Maksim Sisov

Building a Software-Defined Networking System with OpenDaylight Controller

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

30 March 2016

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Maksim Sisov Building a Software-Defined Networking System with OpenDaylight Controller 47 pages + 3 appendices 30 March 2016
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering and Networking Technology
Instructor(s)	Matti Puska, Principal Lecturer

The goal of this project was to create a working and redundant software defined networking system in an isolated networking laboratory. In addition, another goal was to gain experience with the technology and produce well-defined and clear instructions describing how to create the same system.

The project was carried out in a networking laboratory on an ESXi server with virtual machines installed on it. The OpenDaylight controller was chosen as a main controller and Mininet was used as a networking infrastructure orchestrated by the controller.

A usable software defined networking system was built using the OpenDaylight controller and documented. Moreover, a set of instructions with explanations was written. In addition, obstacles and problems were described and a possible solution for the problems was suggested. The system was tested with the VTN application. Overall performance of the system was analyzed.

The results showed that the OpenDaylight controller had been cutting edge technology and an alternative for traditional networks, but some limitations and issues still had existed. Future enhancements were given and a possible practical system for future Metropolia SDN offerings was created.

Keywords	SDN, OpenDaylight, Mininet, VTN, Networking, ESXi

Contents

List of Abbreviations

AAA	Authentication, Authorization, Accounting
ACL	Access Control List
API	Application Programming Interface
ARP	Address Resolution Protocol
CLI	Command Line Interface
DPI	Deep Packet Inspection
FIB	Forwarding Information Base
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
ISP	Internet Service Provider
JVM	Java Virtual Machine
LISP	Locator/ID Separation
MDSE	Model-Driven Software Engineering
NAT	Network Address Translation
NETCONF	Network Configuration
NFV	Network Functions Virtualization
NSP	Network Service Provider
ODL	OpenDaylight
OS	Operating System
OSGi	Open Services Gateway initiative
OVSDB	Open vSwitch Database
QOS	Quality of Service
REST	Representation State Transfer
SAL	Service Adaptation Layer
SDN	Software-Defined Networking
SSL	Secure Sockets Layer
STP	Spanning Tree Protocol
TCP	Transmission Control
TLS	Transport Layer Security
UML	Unified Modelling Language
VM	Virtual Machine

1 Introduction

Computer networks are a complex technology that enables end-devices to communicate with each other. A typical network infrastructure includes routers, switches, servers, web-servers, firewalls, load balancers, intrusion prevention systems and other devices. Efficiency, reliability, flexibility and robustness are the requirements for processing and managing the tremendous amount of data sent over the network. That has made the manufacturing companies of networking devices to implement complex and resource-intensive protocols that enable routers and switches to communicate with each other by packet switching and creating a networking topology for routing purposes. [1, 1-2.]

Generally, the routing and switching solutions can be split into a data and control plane. The data plane is used for packet forwarding between ingress and egress ports according to lookup tables, whereas the local control plane processes a topology and collects a data set to create lookup tables. The high demands on network performance and scalability have made the control plane to be inflexible, over-complicated and hard to operate and manage. [3.]

The solution for this dilemma is taken from the virtualization technology used in data centres for server applications: multiple virtual machines run on a hypervisor are abstracted from physical machine hardware and share the same resources. This paradigm is adopted by Software-Defined Networking (SDN), one of the most revolutionising technologies being a substitution for traditional networks, and an abstraction layer in networking has been introduced. Thus, data and control plane are separated. [1, 3-4.]

The data plane is located in switch hardware, which serves packets according to a forwarding table, whereas the centralized control plane or controller is separately located in other hardware for a centralized device orchestration. This technology enables IT departments, data centres, Network Service Providers (NSP) or Internet Service Providers (ISP) to efficiently use their resources in a more centralized and automated way. [1, 9-11; 3]

Other driving forces for SDN were that the academic community needed a large scale “sandbox” and data centres required the provisioning to be fast to meet today’s requirements. First, the separated data and control planes enabled the researchers to conduct

experiments without violating other layers and traffic – this was a reason why OpenFlow protocol was developed. Secondly, as long as current data centres are moving to cloud technologies, the underlying networking infrastructure should be able to change quickly. Thus, SDN has been an excellent solution for everything mentioned above. [3.]

The aim of this project is to gain experience with the SDN technology and evaluate its status by building a full SDN system using the OpenDaylight controller being the control plane and Mininet being the data plane, and give instructions how to create the SDN based networks. As an additional goal, the technology will be evaluated in terms of usability for lab exercises at the Helsinki Metropolia University of Applied Sciences.

The communication between the controller and the networking devices will be established using a southbound protocol called OpenFlow. As a result, the system should have a connectivity, redundancy and an example application running on top of it.

2 The Software-Defined Networking

This chapter describes the SDN architecture and its components. In addition, data and control planes will be covered separately for a better understanding. To understand how communication between each layer works, northbound and southbound interfaces will be described. In the end, I will provide an overview of the existing SDN controllers and give an introduction to the OpenDaylight (ODL) controller.

2.1 SDN Architecture and Components

Software-Defined Networking is an emerging paradigm that enables network innovation based on four fundamental principles: (1) network control and forwarding planes are clearly decoupled, (2) forwarding decisions are flow-based instead of destination-based, (3) the network forwarding logic is abstracted from hardware to a programmable software layer, and (4) an element, called a controller, is introduced to coordinate network-wide forwarding decisions. [2,207.]

The SDN architecture (figure 1) is based on a principle of separation of the control plane or the network plane from the forwarding hardware and a logical centralization of a control program or a controller that makes forwarding decisions and installs rules on switches or routers. [4,451.] These rules make the forwarding hardware to switch packets between ports. According to this architecture, the SDN can logically be represented as a three-layered architecture:

- Infrastructure layer: This layer is often referred to as a data plane. It involves forwarding hardware, for example, switches and routers, including forwarding components, and Application Programming Interfaces (API).

- Control layer: The other name is a control plane. Network intelligence in a form of logically centralized and software-based SDN controller installed on any UNIX based Operating System (OS) run on any hardware. The control layer manages forwarding hardware and installs forwarding rules via APIs.

- Application layer: Applications and services take control over control and infrastructure layer via Representational state transfer (REST) APIs. The SDN concept enables developers to easily develop applications that perform networking

function tasks. Applications are usually deployed to separate computers or clouds. [4,451-452;5;6,6-7.]

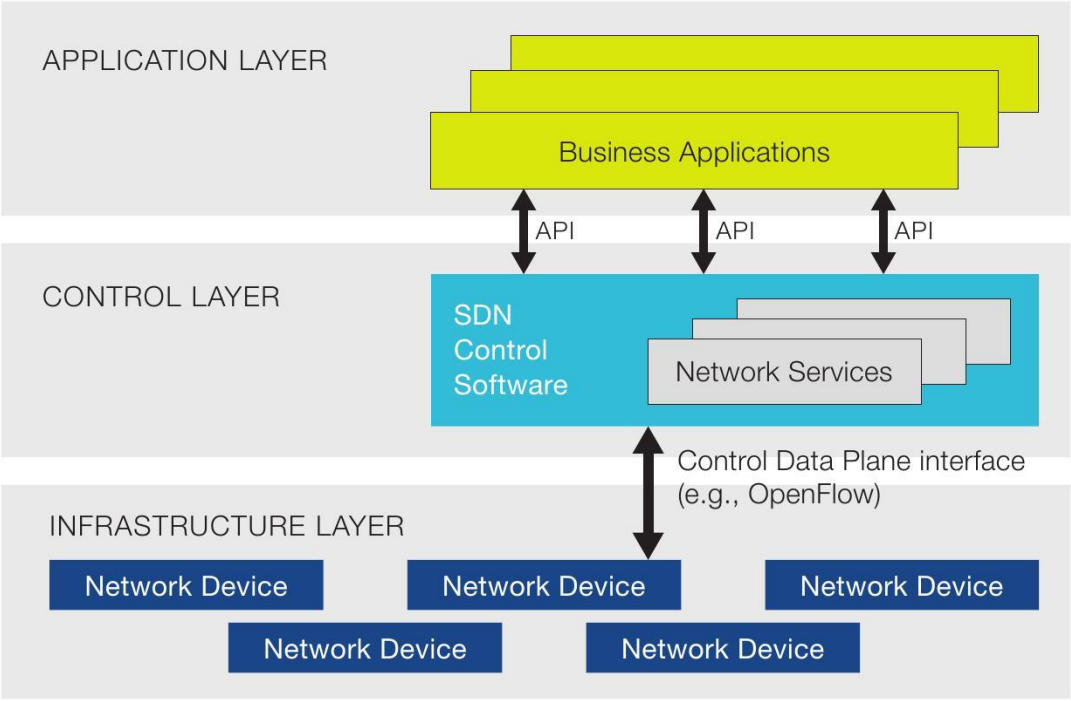

Figure 1. SDN Architecture Copied from SDX Central [3].

The APIs in the SDN architecture are often called northbound and southbound interfaces. Those are used for the communication between hardware, controllers and applications. A southbound interface is defined as the connection between networking devices and controllers, whereas the northbound interface is the connection between applications and the controllers. [5.]

2.2 Data Plane

The data plane infrastructure consists of networking devices. A network device is an entity that receives packets on its ports and performs one or more network functions on them. For example, a received packet can be forwarded, or dropped, or its header can be altered. The packet forwarding function is based on a Forwarding Information Base (FIB) table or tables, which are located on a forwarding hardware and contain MAC addresses mapped to ports, preprogrammed by the control plane. [6,5-6.]

In contrast, the SDN technology uses flow tables, which are not the same as FIB tables. The FIB table is a simple set of instructions based on destination-based switching, whereas the flow table includes a sequential set of instructions and actions [12,15-16]. The flow tables will be covered in more detail in section 2.4.

Sometimes, the data plane is referred to as the fast path for packet management because it requires no further investigation other than an address extraction of the packet destination relying on preprogramed FIB. But one exception exists in the above mentioned process if the packet destination is not found from the lookup tables. In order to proceed, the detected packet with an unknown address is sent to the control plane where the controller makes a forwarding decision using the Routing Information Base (RIB) table, which contains topology, network destinations and metrics associated with routes. In case of SDN, it resides on a software-based controller. [1,16-17.]

FIB tables can reside in a number of forwarding targets - software, hardware-accelerated software (Graphics Processing Unit (GPU)/Central Processing Unit (CPU), as exemplified by Intel or ARM), commodity silicon (Network Processing Unit (NPU), as exemplified by Broadcom, Intel, or Marvell, in the Ethernet switch market), Field-Programmable Gate Array (FPGA) and specialized silicon (Application-Specific Integrated Circuits (ASIC) like the Juniper Trio), or any combination - depending on the network element design [1,16].

In addition to forwarding decisions, the data plane may include other small features and services usually mentioned as forwarding features. Depending on a system, a separate discrete tables may exist for these features or they can act as extensions to the forwarding tables. [1,17.]

For example, the following features can be used in SDN as well as in traditional routing:

- Access Control List (ACL): Can specify drop, alter, pass and other actions for a specific matching flow.

- Quality of Service (QoS): Mapping a flow to a queue on an egress to normalize and provide an uninterruptible service. It may specify packets to be dropped regardless of the forwarding policy.

Figure 2 shows examples of features available on a traditional router and that can be used in SDN.

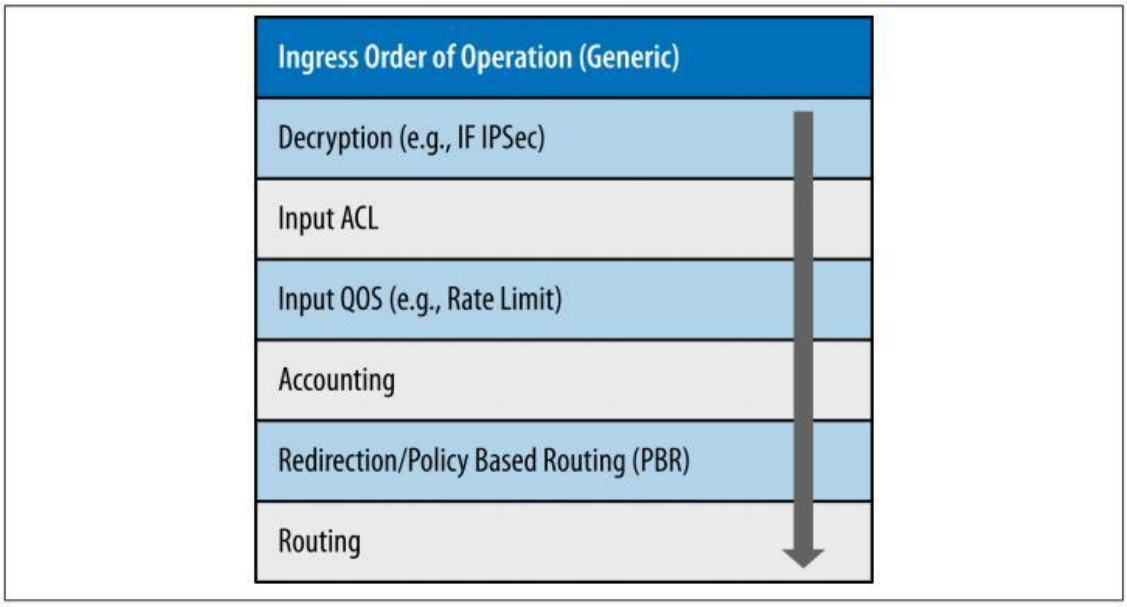

Figure 2 Generic example of ingress feature application on a traditional router. Reprinted from Thomas Nadeu and Ken Grey (2013) [1, 18].

One technology that provides networking functions in SDN, which were removed from the data plane, is called Networking Functions Virtualization (NFV). It is a networking concept that allows to virtualize entire classes of networking functions into building blocks. [33.] Those can be connected or chained in order to create networking services. Examples of the services are Network Address Translation (NAT), Authentication, Authorization and Accounting (AAA) and Secure Sockets Layer (SSL) protocol. A system example of SDN, NFV and clouds can be OpenDaylight controller with OpenStack cloud technology. This combination provides fast provisioning, advanced NFV features and scalability.

Referring to hardware, there are white box switches which exist for SDN purposes. They represent foundational elements of the networking infrastructure that enable organizations to choose and use only those features they need to realize as their SDN objectives.

The white box switches can come with pre-installed OS with minimal software installed or be sold as a bare metal device. This allows users and businesses to customize switches according to their needs. [34.]

2.3 Southbound Interfaces

The APIs of the Southbound interface enable an SDN controller to efficiently control a networking infrastructure and make dynamic changes in accordance with real-time traffic, its demands and needs [35].

Several examples of southbound APIs exist:

- OpenFlow: An SDN southbound protocol proposed to be a standard protocol in industry and used to transport a control logic function from a switch to a controller.
- Open vSwitch DataBase Protocol (OVSDB): Management protocol used by Open vSwitch open source software switch.
- Locator ID Separation Protocol (LISP): It provides a flexible map-and-encap framework that can be used for overlay network applications, such as data centre network virtualization, and Network Function Virtualization (NFV) [8].
- Network Configuration Protocol (NETCONF): The protocol is used for networking devices configuration.

Other examples are shown in figure 3 illustrating APIs supported by the OpenDaylight controller.

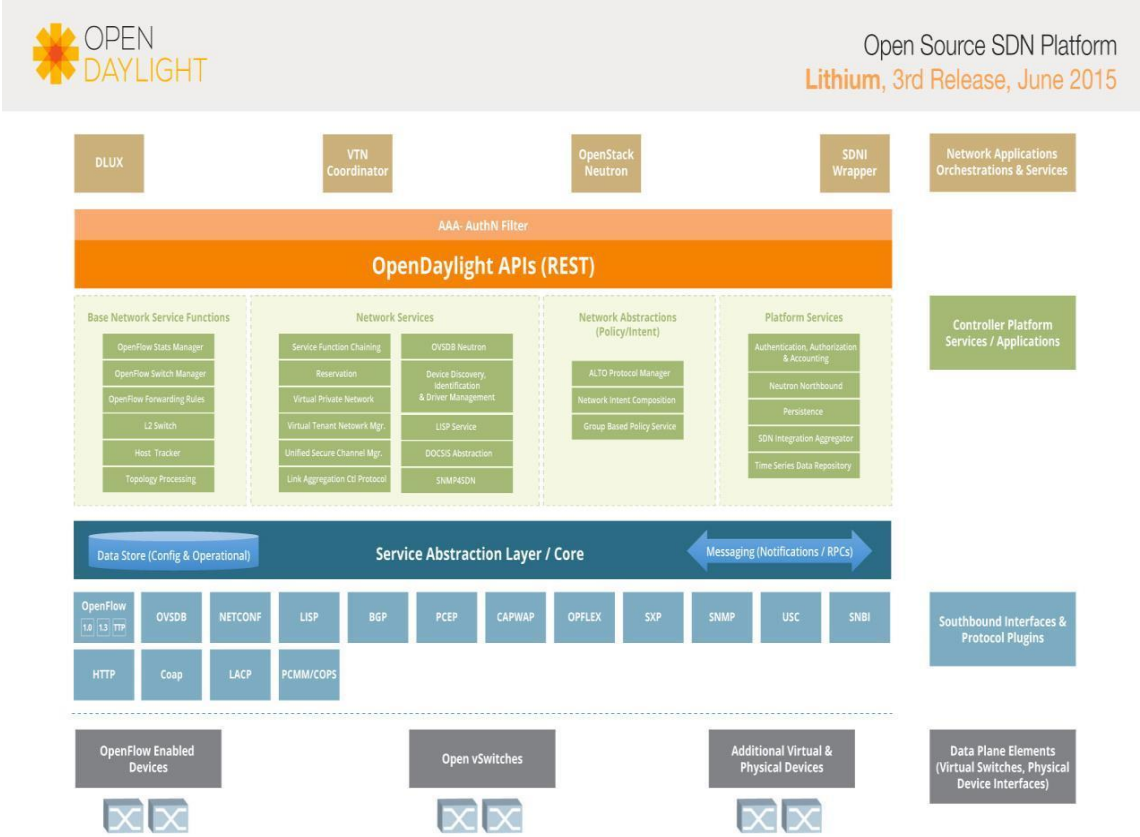

Figure 3 OpenDaylight Architecture and Supported Southbound APIs. Copied from [3].

The OpenDaylight controller supports many protocols to be used by different users, businesses and vendors.

2.4 OpenFlow Protocol

Communication between an SDN controller and networking devices is handled through the southbound APIs. One of the most popular protocols is OpenFlow. It was first proposed as a way for researchers to conduct experiments in production networks. However, its advantages led to it being used beyond research. Due to its flexibility, the protocol provides a flexible routing of network flows without interrupting other traffic. This possibility was achieved by separating the data and control plane, where a controller orchestrates network traffic and makes or changes rules in networking devices.

OpenFlow does not require any hardware changes and vendors do not have to open their systems to support it as long as only a few well-known interfaces have to be open for the control plane. The Ethernet switches just need some software changes that can

be provided as a firmware extension for the existing hardware. The reason why it is possible is the fact that all the modern switches and routers contain flow tables that are used for firewalls, Quality of Service (QoS), NAT, and for statistics collection. Although, the implementation of the flow tables may vary from one vendor to another, there is a common set of functions that run on many switches and routers. The OpenFlow protocol exploits them. [9,1; 2,208-209.]

As mentioned above, using the OpenFlow protocol, a controller is able to not only set forwarding rules by sending flow entries, but learn the network statistics, hardware details, ports, connectivity status and the network topology of Ethernet switches.

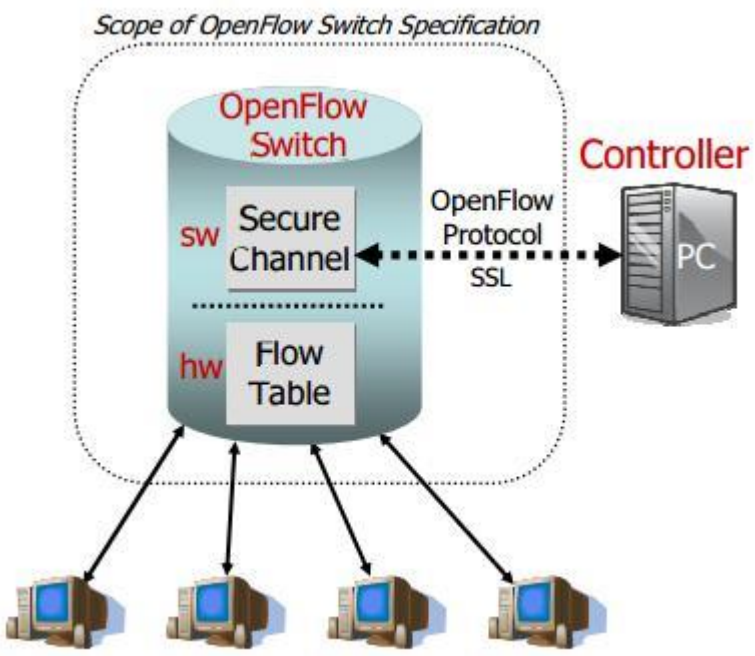

Figure 4 OpenFlow switch specs. Copied from [46].

As shown in figure 4, an OpenFlow Switch consists of at least three parts: (1) a Flow Table, with an action associated with each flow entry, to tell the switch how to process the flows, (2) a Secure Channel that connects the switch to a remote control process (called the controller), allowing commands and packets to be sent between a controller and the switch using (3) the OpenFlow Protocol, which provides an open and standard way for a controller to communicate with a switch [11,3-4].

Flow tables and group tables consist of flow entries that define how switches should behave with a traffic flow coming from/to different physical and virtual interfaces. The tables of an OpenFlow switch are numbered starting from 0. First, incoming packets are

matched against flow entries in table 0. When a flow entry is found, the instructions included in that entry will be executed against the packet. The instructions may explicitly send the packet to another flow table, repeating the process again and again. A flow entry can only direct the packet forward, not backwards (see figure 5), increasing the numbering of the tables. When the packet processing pipeline stops, the packet will be processed in accordance with the instructions set matching this packet. [12,13-18.]

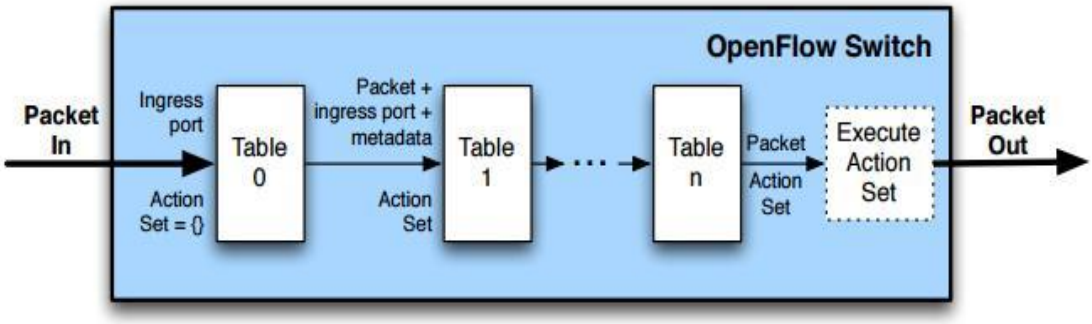

Figure 5 Packets match against tables. Copied from [46].

If a flow table for a packet is missing, it is called a table miss. There are several instructions that can be set for this situation. [12,17.] Options include dropping the packet, passing it to another flow-table or sending it to a controller over the control channel as shown in figure 6.

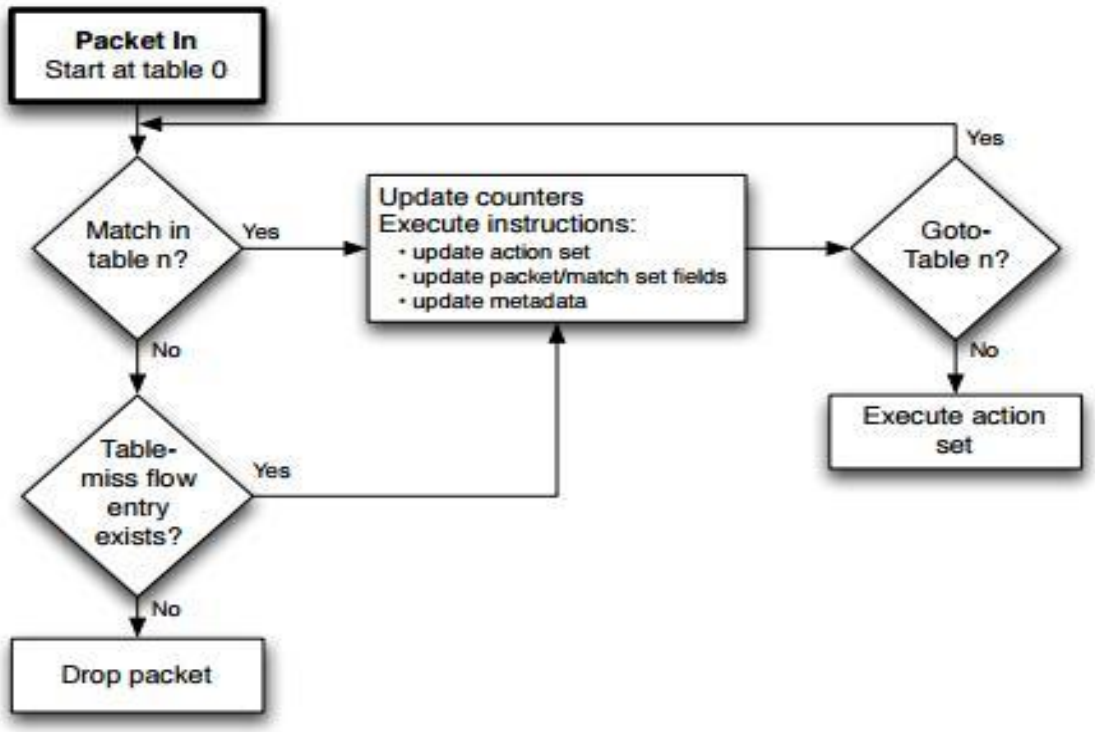

Figure 6 Packets Flowchart. Copied from [12].

The flow entries in the flow tables have the following structure (see figure 7):

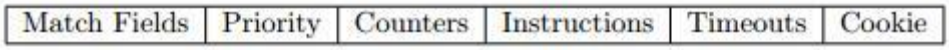

Figure 7 Main components of a flow entry in a flow table. Reprinted from [12].

Explanation of the fields is the following:

- **Match Fields**: Include ingress port and L2-L4 packet headers. As an option, metadata can be specified by a previous table.
- **Priority**: Contains flow entry precedence.
- **Counters**: Increased if there is a packet match.
- **Instructions**: Used for action set modification
- **Timeouts**: Maximum time when flow is expired by the switch
- **Cookie**: Opaque data value chosen by the controller. May be used by the controller to filter flow statistics, flow modification and flow deletion. Not used when processing packets [12,15].

OpenFlow Channel is the interface that is used for a connection between each OpenFlow switch and a controller. Using this interface, the controller can receive switch generated events, send packets out of the switch and, last but not least, configure and manage the switch. The connection between the switch and the controller is encrypted with Transport Layer Security (TLS) protocol. However, it may be run directly over Transmission Control Protocol (TCP). [12,26.]

2.5 Control Plane

The responsibility of the control plane is to configure and manage the data plane devices over southbound interfaces and instruct the networking devices how to handle packets. It is usually distributed and can involve several controllers at the same time, which is called clustering. [1,5;1,9;1,11.] Examples of the controllers that reside on the control plane are OpenDaylight, Floodlight, OpenContrail and FlowVisor.
The main functionalities of the control plane are

- Maintenance and discovery of topology
- Instantiation and selection of a packet route
- Mechanisms for path failover.

Moreover, the plane also provides services for applications to use the data plane and data gathered by the control plane to provide other functions within the network. [1,11-15.] Examples of the applications include network provisioning, advanced network topology discovery, path reservation and firewall. Communication between the control plane and the applications is established through Northbound APIs.

2.6 Southbound Interfaces

The role of the northbound interface is to provide a high-level API between the controller and applications that can compute network operations based on the events collected by the control plane. Figure 8 represents the existing APIs in the SDN system.

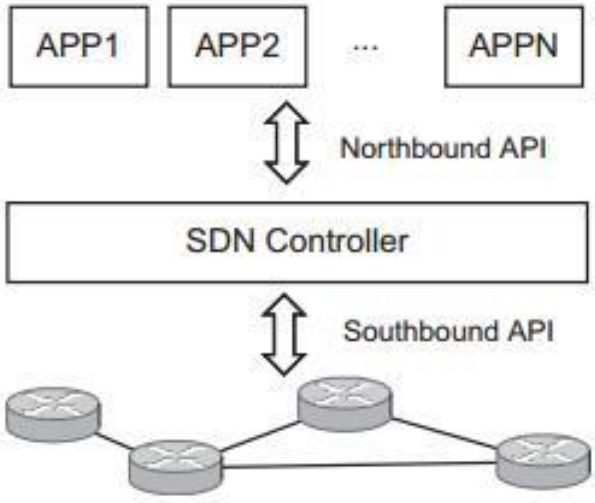

Figure 8 the architecture of Software-Defined Network. Copied from Li Li (2014) [17,1].

Orchestration platforms, such as OpenStack, and automation stack, such as Puppet or CFEngine, also use the SDN northbound APIs to integrate with the system. The main goal of the APIs is to abstract the underlying network infrastructure and enable application developers to have their attention on the application development instead of understanding how their changes can affect the inner-working of the network. [13.]

The main architectural style for northbound API is Representational State Transfer (REST). The REST APIs support flexibility and can provide different functions at the same time and make changes to those functions without interrupting clients. This is achieved by the mechanism called hypertext driven navigation of the connected resources.

In particular, a REST API consists of the connected REST resources that provide different services through uniform interfaces. In the case of SDN, it includes services from both data and control planes, such as switches, routers, subnets, networks, NAT devices, and controllers. [1,46.]

The example of the REST GET request looks like in listing 1:

```
controller> curl -u admin:admin -H 'Accept: application/xml'
'http://<controller-ip>:8080/con-troller/nb/v2/flowprogram-
mer/default'
```

Listing 1. Example of the REST GET request.

The output of this query shows installed flows in networking devices.

2.7 Overview of the Existing SDN Controllers

SDN controllers reside on the control plane. They are the “brains” of the network and perform management functions over switches or routers via southbound APIs. There are several open source controllers available for usage, for example, OpenDaylight, Floodlight, OpenContrail and FlowVisor.

In this project, the OpenDaylight controller was used to create an SDN system. The reason why I chose this controller was that it is supported by Linux Foundation and the project is always evolving. Secondly, there are platinum contributors such as Cisco, Ericsson, Brocade, Intel and others which support and develop the project which makes it reliable for usage. Furthermore, it allows other non-OpenFlow protocols to be used and provides such functions as clustering, multilayer network optimization, load balancing and others. Last, but not least is that the OpenDaylight controller has well-structured documentation pages, which helps in the case of using such a complex system. Thus, the OpenDaylight will be covered in sections 2.8, 3 and 3.1-3.5.

2.8 Introduction to OpenDaylight Controller

The OpenDaylight controller is an open-source controller under the Linux Foundation collaborative projects. It is a software application and as a Java Virtual Machine (JVM) it can be run on any OS that supports Java. As shown in figure 9, it consists of several blocks: Northbound REST APIs, Network Service Functions, Network Orchestration Functions, Service Abstraction Layer and Southbound APIs.

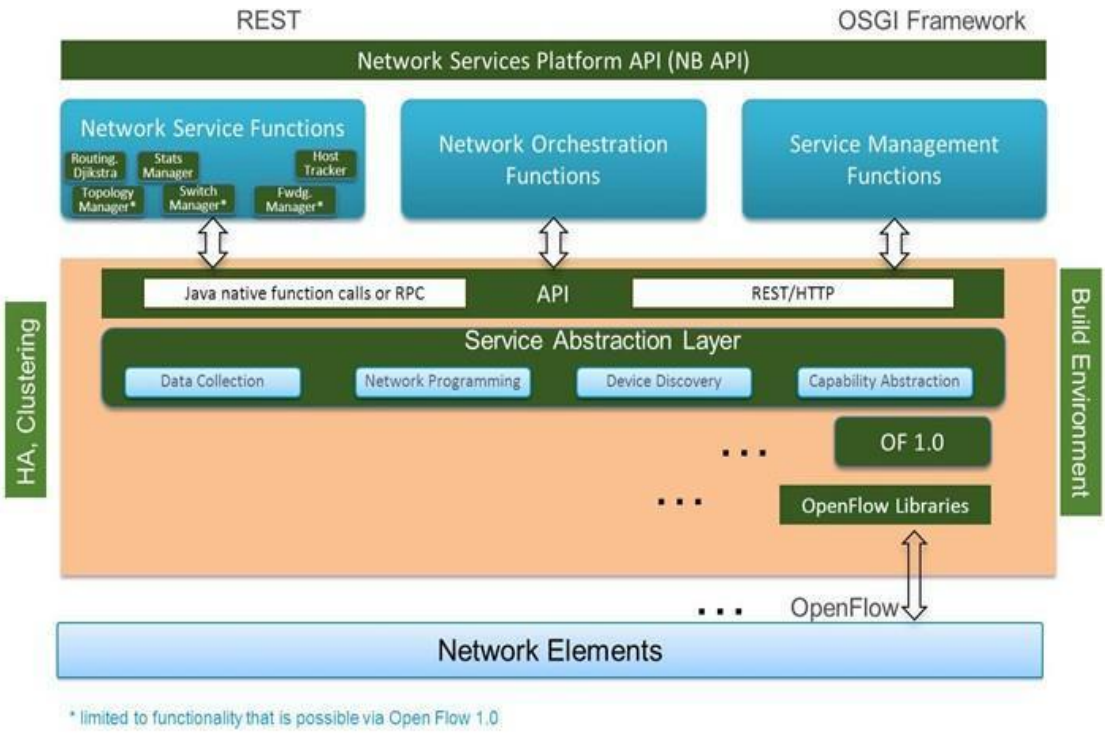

Figure 9 Structure of the OpenDaylight Controller. Copied from [3].

As a southbound protocol, the OpenDaylight can use, for example, OpenFlow 1.0-1.3 and BGP-LS protocols. [21.]

The newest controller version (Lithium at the time of writing) is designed using Model-Driven Software Engineering (MDSE) by Object Management Group (OMG). MDSE describes a framework based on consistent relationships between (different) models, standardized mappings and patterns that enable model generation and, by extension, code/API generation from models. This generalization can overlay any specific modeming language. Although OMG focus their solution on Unified Modelling Language (UML), YANG has emerged as the data modelling language for the networking domain. [18, 2.]

There are open northbound REST APIs available for application usage. Open Service Gateway Interface (OSGi) framework is also available for applications. This framework is used for running the applications in the same address space as the controller, whereas the REST APIs are used by web applications that do not reside in the same address space (or hardware) as the controller. [18,3-4.]

Service Abstraction Layer (SAL) was chosen as the modular design of the OpenDaylight controller (figure 10). It enables the software to support multiple protocols on the south-bound and provides coherent services for modules and applications.

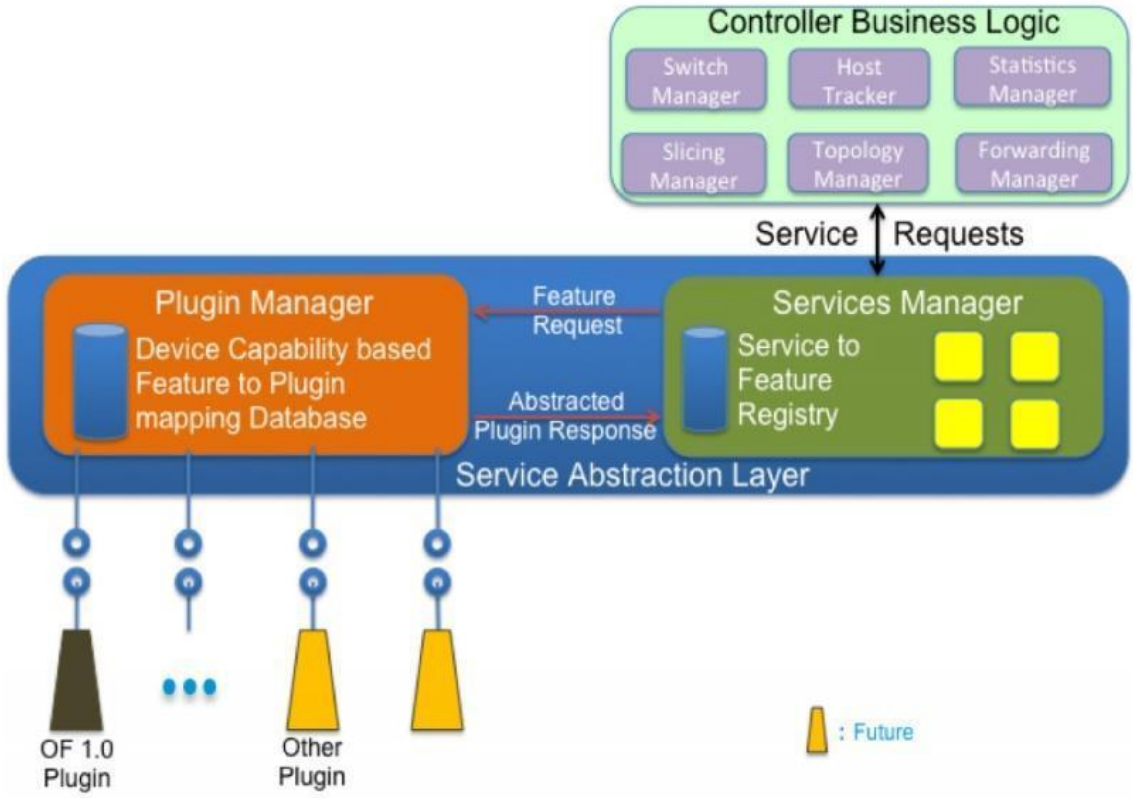

Figure 10 Service Abstraction Layer. Copied from [47].

While the OSGi framework allows a dynamic linking for different southbound protocols, services such as Device Discovery being used by modules like Topology Manager are provided by the SAL. As long as services are constructed using the features exposed by the plugins, the SAL can map to the appropriate plugin and use the most appropriate southbound protocol for networking device interaction based on the service request. [25.] Figure 11 represents the previous mentioned plugin linking.

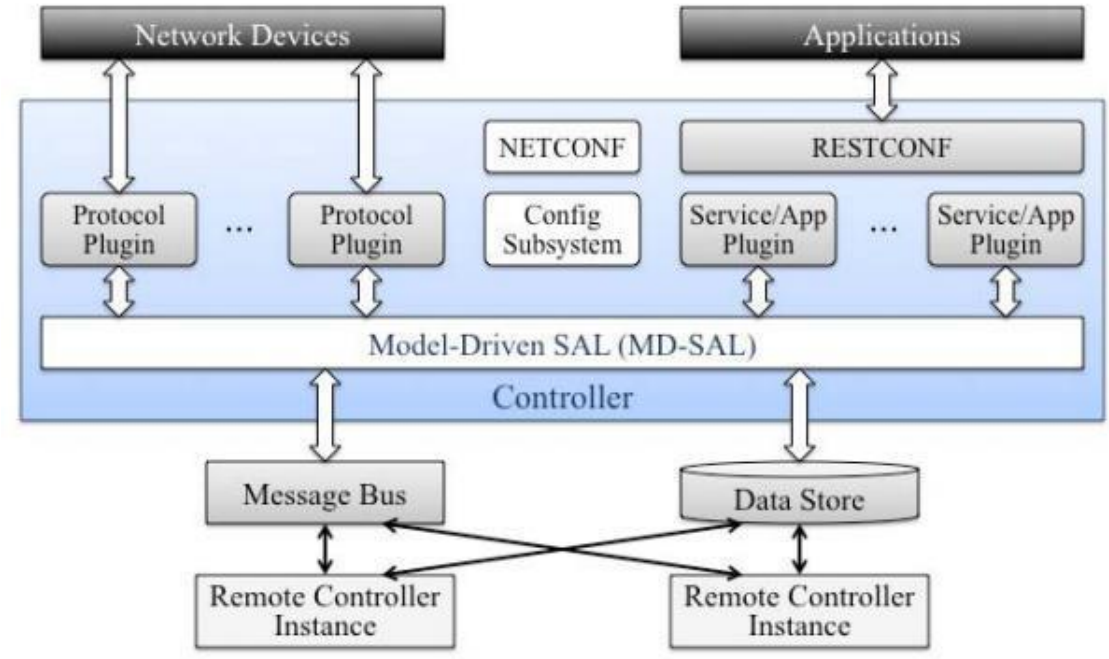

Figure 11 OpenDaylight Controller architecture. Reprinted from Medvedev (2014) [18, 3].

The above mentioned architecture enables the controller to be

- Flexible: It is able to serve a vast diversity of applications using the same framework and programming model and provide reliable APIs.
- Scalable in the development process: There is no common infrastructure subsystem. Instead, plugins are developed independently of each other. Thus, the controller has short integration times.
- Run-time extensible: The controller can easily adapt to new data models and dynamically load other plugins. [18,3-5;25;36.]

As regards scalability, the OpenDaylight controller has over 17 projects being developed at the same time, including, for example, L2 Switch, OpenDaylight User eXperience (DLUX), and Virtual Tenant Network (VTN). In addition, the OpenDaylight controller supports the High Availability model (HA). There can be several instances of the controller run that act as one logical controller. Thus, reliability and redundancy can be achieved.

3 SDN Setup with OpenDaylight Controller

In this section I will describe in detail how to set up virtual machines in a virtual environment, configure virtual network between the virtual machines and install the OpenDaylight controller along with Mininet for testing purposes. The goal of this part is to guide the reader through the installation and configuration steps of the controller.

3.1 Environment Requirements

The SDN system was built in the virtual environment with several Virtual Machines (VM). I was provided with an access to the DELL R710 server with VMware ESXi 5.5.0 installed. The server was run with 8 core 2.40 GHz CPU and 90 GB of RAM. Of course, the system did not require that much of computing power, and, thus, virtual machines were set up accordingly.

Ubuntu Server 14.04.3 LTS 64-bit was chosen as a guest OS. There was no specific reason behind of this choice, except the fact that I was mostly familiar with that one and felt comfortable with it.

For the project, I used 5 VMs: 3 for the OpenDaylight controllers to be run as a cluster and redundancy, 1 for SDN applications and yet another one for Mininet – a virtualized network. There was no specific need to have separate VMs for applications and Mininet, but I decided to have them separated for a better resource usage.

The configurations for the VMs were taken from an Ericsson SDN laboratory provided for the OpenDaylight community and were adjusted in such a way to run the system smoothly without any resource troubles [43].

The OpenDaylight controllers will have the following configuration:

- 6 Cores
- 8 GB RAM
- 32 GB HDD
- 2 virtual network adapters (1 adapter for Secure Shell (SSH) connection to the VM and 1 for intercommunication between the controllers and Mininet).

Another VM with Mininet will have another configuration:

- 4 Cores
- 6 GB RAM
- 12 GB HDD
- 3 virtual network adapters (1 adapter for SSH connection to the VM, 1 for inter-communication with the controllers and 1 spare in case of connection with other virtual devices such as OpenvSwitch).

The last instance for the applications will have the following configuration:

- 4 Cores
- 6 GB RAM
- 32 GB HDD
- 2 virtual network adapters (1 adapter for SSH connection to the VM and 1 to communicate with the controllers).

Of course, the RAM and HDD configuration depends on the number of applications and features to be installed.

I decided to have the VMs' network to be configured with different network addresses for different purposes – 10.94.159.100-105 for the Internet and SSH connection, 192.168.1.0/24 for the communication between the controllers and underlying Mininet network as shown in figure 12. I wanted the traffic not to overlap with another one and hide the controllers from the outside infrastructure network.

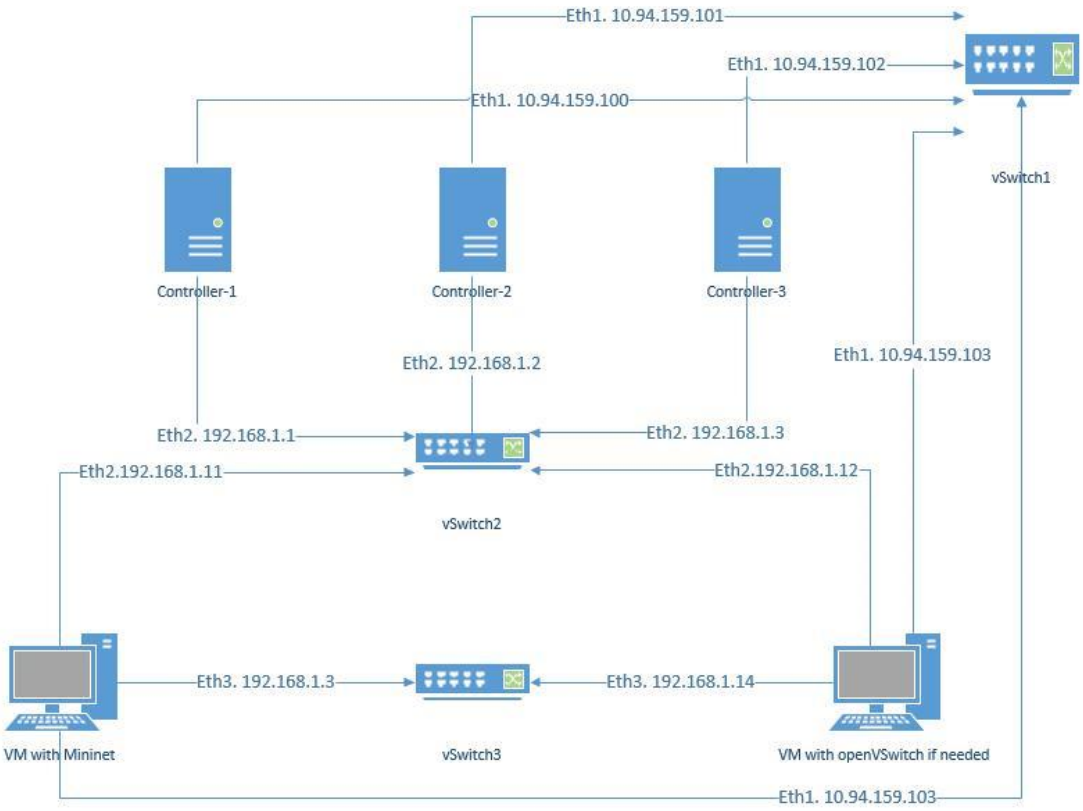

Figure 12 System topology.

As soon as I made a decision about the networking topology, I started to configure my VMs.

3.2 Virtual Machines Setup

I had access to the local ESXi server that was running my virtual machines. In order to connect to that, vSphere client was needed. At the time of writing, vSphere version 5.5.0 was installed on the Metropolia laboratory systems, even though version 6.0 was available.

As long as the goal of the project has been to gain experience and provide instructions, the configuration steps will be given without the name and value parameters because they can differ from system to system to be configured.

After connecting to the ESXi host, I created 5 VMs – 3 for the OpenDaylight Controller, one for the SDN applications and one for Mininet. While creating virtual machines, it was important to remember that I decided to have separate networking adapters for different

purposes as was described in the previous subchapter. Thus, I needed separate vSwitch[1] instances provided by the ESXi server. One adapter was connected to the vSwitch that had been mapped with a physical adapter that had an Internet connection and another adapter was connected to the vSwitch that routed internal traffic only.

To start with, create 3 VMs for controllers, 1 VM for applications and 1 VM for the Mininet with previously given system requirements if following my configurations.

The topology in the VMware ESXi should look like shown in the figure 13.

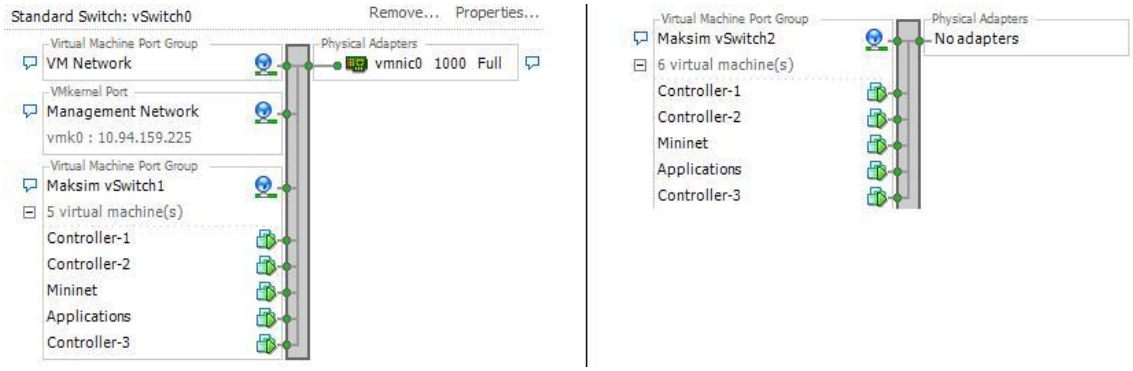

Figure 13 Virtual Machines Topology from ESXi

The preparation step is completed. The next phase is to install an OS.

Configuring guest OS in Virtual Machines

Each VM instance will run the same OS – Ubuntu Server 14.03.3 LTS 64-bit. Hence, installation and configuration steps are the same for each VM. As previously said, I chose to use the Ubuntu Server because I felt more comfortable with it.

To continue the guidance, start with supplying the machine with an OS image to be installed. Once done, start the machine and configure the OS. The step is straightforward and does not require any special configurations except the *"OpenSSH server"* package that should be installed for SSH connectivity. Of course, it is possible to connect to the VMs through the console in the ESXi host, but I wanted to have a remote connectivity to connect from any remote place.

[1] vSwitches were created by the server administrator and did not require any additional configurations.

As soon as the Ubuntu server boots up, the network interfaces should be configured. Each VM will have different IPs and interfaces. My configurations can be found in appendix 1. In order to save the configurations and not let the interfaces to be erased after reboot, configure them in the *"interfaces"* file located in system root *"/etc/network/"* folder.

After this step, there is no need to open a console connection through vSphere client program to connect to the VMs any more. It is now possible to establish an SSH connection from any computer in the same network or from any remote place if SSH connection is allowed. I will not go through this step as long as it is out of the scope of this project.

At this stage, the virtual machines setup is completed. Repeat previous steps to configure other VMs in accordance to the system requirements.

3.3 Installing and Configuring OpenDaylight

As soon as all the VMs are up and running, it is a time to start the OpenDaylight controller installation. In order to verify the controller is running, the Mininet network will be installed.

First, connect to the first VM via SSH. To make other steps simpler, create a *"Downloads"* and download a binary Lithium version (the latest version at the time of writing) of the controller from https://www.opendaylight.org/downloads to the directory that was created. The simplest way to do that is by *"wget [OPTION]... [URL]..."*. Extract the downloaded archive to *"home"* directory and name it *"ODL"*. It is possible to download the source code and compile it as well, but it might not be stable and it might have some overlapping or broken features.

Configuring the OpenDaylight Controller

Before the controller can be started, it is wise to change its IP address. The controller will be accessible using that address and isolated from the external network. By default, the controller listens to the loopback address and is not accessible from other VMs.

To configure the IP address, proceed with the following steps:

- In the *"ODL/etc"* folder open *"custom.properties"* file.
- Replace the IP address with the address configured to the second interface of the OS in the following fields: *"netconf.tcp.address=", "netconf.tcp.client.address=" and "of.address="*[2].

Next, start the controller by executing *"./karaf",* which is a binary file that starts the controller, in the *"ODL/bin/"* directory. The controller will start and will be ready for work. [37.]

L2Switch Features Installation Steps

At the moment, the controller is not capable to do any networking-related tasks. Thus, L2Switch features should be installed. The Command Line Interface (CLI) command looks like the following:

```
karaf@root>feature:install odl-l2switch-switch-ui
```

Verify that the components are properly installed:

```
karaf@root>feature:list -i | grep l2switch
```

As shown in figure 14, along with the "*odl-l2switch-switch-ui"* feature*,* the *"odl-l2switch-hosttracker", "odl-l2switch-addresstracker", "odl-l2switch-arphandler", "odl-l2switch-loopremover" and "odl-l2switch-packethandler"* should be automatically installed as well.

[2] I used the following addresses in my system: controller-1 – 192.168.1.1, controller-2 – 192.168.1.2 and controller-3 – 192.168.1.3.

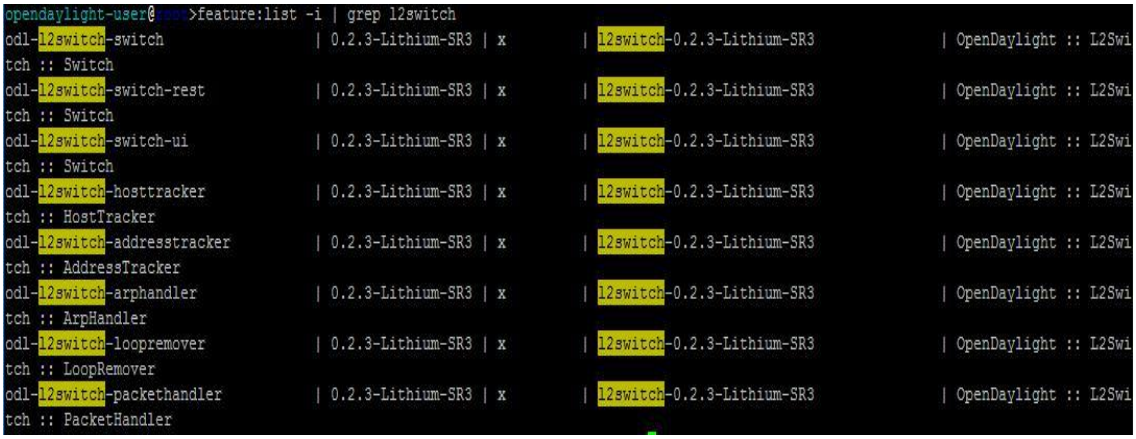

Figure 14 Installed L2Switch Components.

The following architectural components play different roles in the L2Switch project:

- **Packet handler**: Is responsible for packets decoding and dispatching decoded packets notification.
- **Loop remover**: Removes loops in the network and updates Spanning Tree Protocol (STP) status of the network ports.
- **Host tracker**: Tracks the hosts and updates operational topology tree.
- **Arp handler**: Handles the decoded Address Resolution Protocol (ARP) packets, either by installing proactive flood flows or by dispatching packets back to network, based on the configuration.
- **L2switch Main**: Is responsible for flows installation on each switch, based on addresses learned and network traffic.

The above mentioned components provide Layer 2 switch connectivity. What it means is when a packet comes from a networking device, L2Switch will learn about the MAC address of a source device. If the destination is unknown, it will send a broadcast message on all external ports in the network and expect to receive a reply message from the device the packet should be sent to. Otherwise, if it knows about the destination, the packet will be forwarded to the destination. [38; 39.]

Installing DLUX Features

The DLUX is pronounced like *"Deluxe".* It is responsible for the Graphical User Interface (GUI) features in the OpenDaylight controller. It can be installed with the following commands:

- Execute and install "`feature:install odl-restconf`" in the OpenDaylight console,
- Next, install "`feature:install odl-mdsal-apidocs`",
- Finally, execute "`feature:install odl-dlux-all`" command.

The *"odl-restconf"* provides REST APIs for applications and supports *"Options", "GET", "PUT", "DELETE"* HyperText Transfer Protocol (HTTP) commands. The *"odl-mdsal--apidocs"* provides apidocs explorer and lists all the available APIs on the controller. The *"odl-dlux-all"* installs the GUI features for the OpenDaylight controller. [14.]

In order to verify that the above mentioned features are installed, execute "`feature:list –i | grep [MODULE_NAME]`".

The controller is ready to provide basic services to networking devices. In order to test them, the Mininet network will be installed on another VM. Repeat the same steps for the rest controllers.

3.4 Installing and Configuring Mininet

In brief, the Mininet is a network emulator. It can create a network of virtual switches, controllers, servers, hosts, links and other network infrastructure devices to work with. It uses OpenFlow switches as well as openvSwitches for the network configuration. I decided to use the Mininet to test the OpenDaylight controller because it is powerful, supports IP connectivity, and allows to create a huge network on the VM without any special requirements. Moreover, I could not find any real hardware that would support OpenFlow protocol. [40.] Thus, the Mininet was used.

To continue the installation process, it is wise to install the Mininet emulator from source (*github.com/mininet/Mininet)* to get the newest available version (Version 2.2.1 was

available at the time of doing the project). It is possible to download a ready-made Mininet VM from http://mininet.org, but it has additional features that, for example, I have not used and they would have just wasted computing resources. [41.]

In order to install the Mininet, proceed with the following steps:

- Clone the source code "`git clone git://github.com/mininet/Mininet`" to Mininet machine.
- In the *"Mininet'* folder list available versions "`git tag`".
- Choose the version to be installed "`git checkout –b [VERSION]`".
- Once the code is fetched, install Mininet "`Mininet/util/install.sh -a`". The *"-a"* option will install everything included in the Mininet: dependencies to the openvSwitch, OpenFlow wireshark, POX and other packages.
- After the installation is completed, verify the basic functionality "`sudo –mn –test pingall`". This command will test IP connectivity between the hosts in the network created in Mininet. The result should be the same as shown in figure 15. [41.]

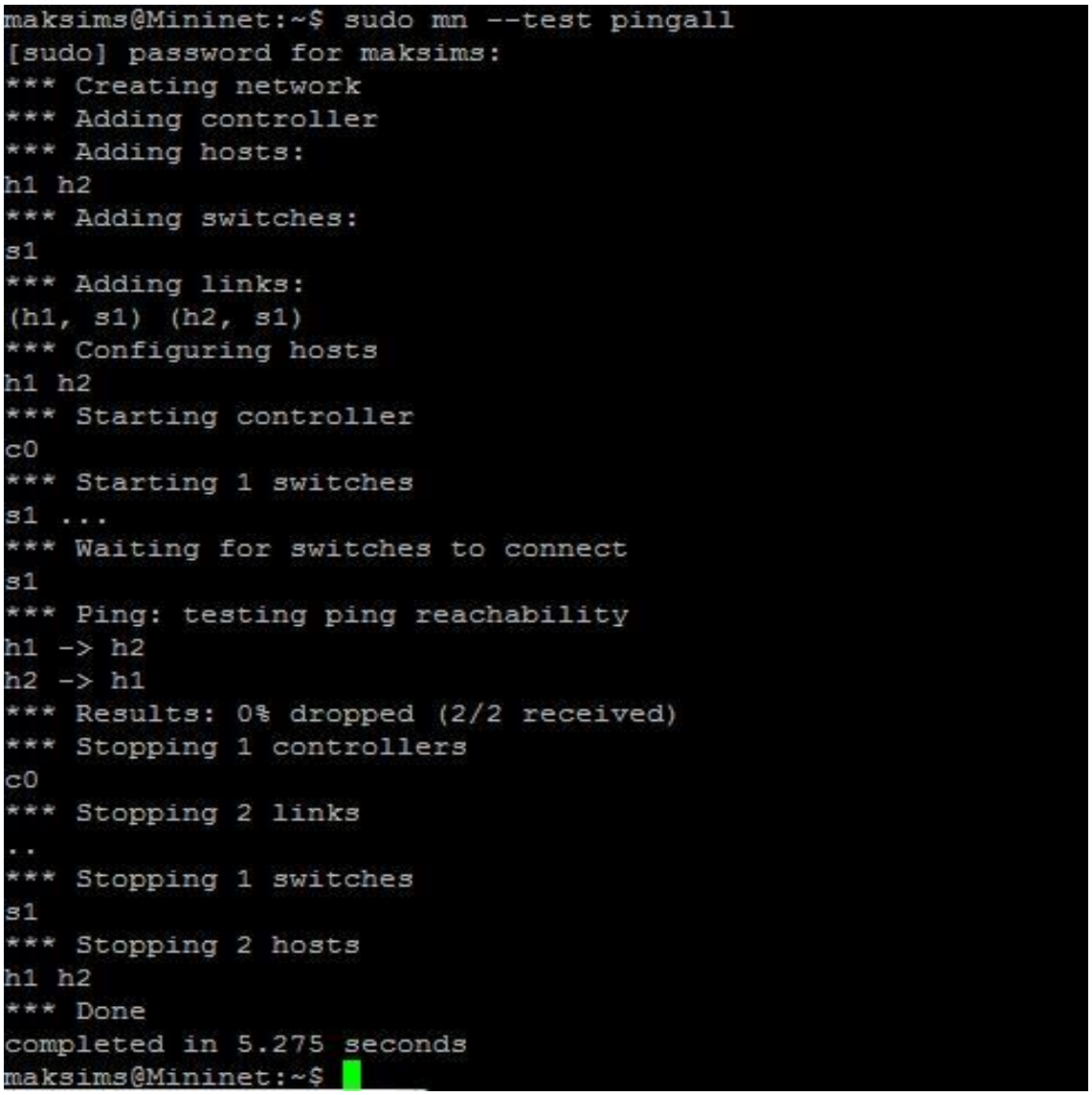

Figure 15 Testing Mininet functionality.

After successful installation and testing, it is now possible to connect to the OpenDaylight controller installed in the Installing and Configuring OpenDaylight section.

For this purpose, execute "`sudo mn --controller=remote,ip=[CONTROLLER_IP] --topo tree, 3`" command using an appropriate IP address. This will create 7 switches and 8 hosts. If the controller is configured right, the "`Mininet>pingall`" command should execute without any failures as shown in figure 16. The result will show that L2 switch features work properly and IP connectivity between hosts in the Mininet is successfully established.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8
h2 -> h1 h3 h4 h5 h6 h7 h8
h3 -> h1 h2 h4 h5 h6 h7 h8
h4 -> h1 h2 h3 h5 h6 h7 h8
h5 -> h1 h2 h3 h4 h6 h7 h8
h6 -> h1 h2 h3 h4 h5 h7 h8
h7 -> h1 h2 h3 h4 h5 h6 h8
h8 -> h1 h2 h3 h4 h5 h6 h7
*** Results: 0% dropped (56/56 received)
mininet>
```

Figure 16 Mininet "pingall" Successfull Execution.

Once the GUI has been installed in section 3.3, it is now possible to check the networking topology created by the L2Switch features.

The access link is "http://[CONTROLLER_IP]:8181/index.html". The default credentials are *"admin"* for both username and password. [14.]

From the left menu choose "*Topology"* and it will appear (see figure 17). Some failures exist because the project is still in the development process. That is why the topology sometimes is not well structured as shown in figure 17.

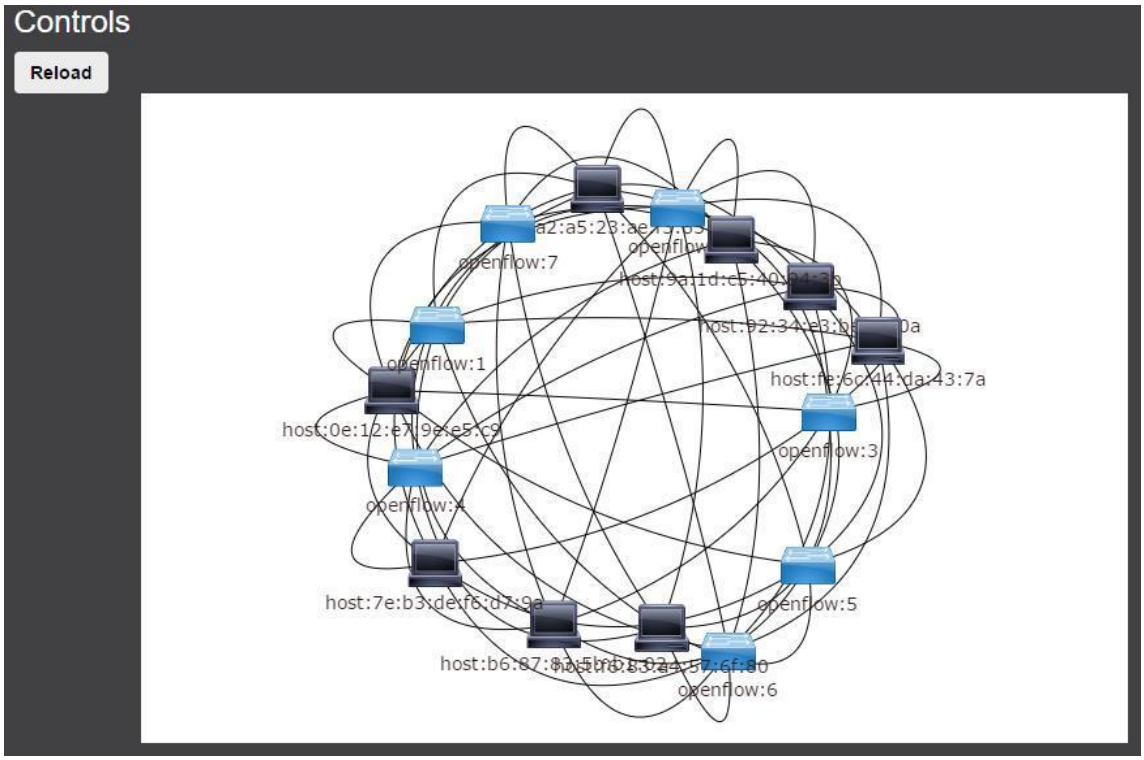

Figure 17 Topology in DLUX.

In order to change the credentials, first, get the user IDs stored: "`curl -u admin:admin http://[ CONTROLLER_IP]:8181/auth/v1/users`". For example, if you want to change password for user "admin", check the user ID for this account as long as it is used for a REST call. Now a json file should be created with right fields.

```
{
"name":"ACCOUNT_NAME",
"description":"ACCOUNT_DESCRIPTION",
"enabled":"TRUE/FALSE",
"email":"",
"password":"NEW_PASSWORD"
}
```

Listing 2. JSON file with credentials. Copied from [19].

Next, a rest call should be made using the created a .json file. [42.]

```
curl -u admin:admin -X PUT -H "Content-Type: application/json"
--data-binary     @./[FILE_NAME].json     http://[CONTROLLER-
IP]:8181/auth/v1/users/[ USER_ID]
```

Listing 3. REST call. Copied from [42].

At this stage, the basic configuration of the controller and the network emulator is ready. In the next section, I will show how to install a cluster of controllers and possible ways of testing.

3.5 Configuring Redundancy: OpenDaylight Clustering

In order to make the controller reliable and achieve the redundancy goal, I decided to use the clustering technology available in OpenDaylight. It is a High Availability (HA) model that relies on Akka[3] components: Akka remoting, Akka clustering and Akka persistence. [24.]

[3] Akka is a toolkit used for building concurrent and distributed message-driven applications on the JVM.

Akka remoting is used for peer-to-peer communication between components in the OpenDaylight project. Akka clustering provides a fault-tolerant and peer-to-peer based cluster service. [22; 23.] It involves nodes, clusters and leaders, where

- nodes are members of the cluster
- cluster is a set of nodes joined though the cluster service
- leader is a node that acts as a leader in the cluster.

Akka persistence enables stateful actors to persist their internal state so that it can be recovered when an actor is started, restarted after a JVM crash or by a supervisor, or migrated in a cluster. The key concept behind Akka persistence is that only changes to an actor's internal state are persisted but never its current state directly (except for optional snapshots). [24.]

The OpenDaylight clustering technology suggests to use at least three controllers for redundancy, because if one is out of order, two of them can still operate and provide reliability and redundancy. [25.]

To run the controllers in a three-node cluster, proceed with the following steps (these steps should be done in each controller):

- Each member of the cluster should have an identifier. In the *"akka.conf"* file located in *"ODL/configuration/initial/"* folder find *"roles = ["member-1"]"* field.
- Replace member names in accordance with the controllers' name. For example, controller-1 will be member-1, controller-2 will be member-2 and controller-3 – member-3.
- Still in the same file, replace the IP address in *"netty.tcp { hostname = "[controller-ip]" .. "* with the actual address of each controller. Note that there are 2 instances of this line in the file.
- Define seed-nodes, which will be part of the cluster, in the field that looks as the following:

```
cluster {
     seed-nodes = ["akka.tcp://opendaylight-cluster-
data@[controller-ip]:2550"]
```

Listing 4. Seed-nodes set up.

- Set the IP addresses in accordance with the cluster topology and save the file. (Note that there will be one more place under *"odl-cluster-rpc"* where changes should be made).
- The last step is to define shard[4] replications that can be found in the *"ODL/configuration/initial/module-shards.conf"* file. Set each *"replicas"* names to match the *"role"* names in the hosts *"akka.conf"* file.

As soon as the configuration is done, start each instance of the controller in the different VM – *"./ODL/bin/karaf/"* and proceed with the following commands

- Install restconf "feature:install odl-restconf-all" that provides REST APIs of the controller.
- Add the clustering feature "feature:install odl-mdsal-clustering" that involves source code of akka based features for HA.
- Install Jolokia "feature:install http" "bundle:install –s mvn:org.jolokia/jolokia-osgi/1.1.5", which is a remote Java Management Extension (JMX) with JSON over HTTP [26].

The controllers will automatically start to communicate with each other and after a couple of minutes the connection will be established. [25.]

Testing the Cluster

The cluster can be tested using two tools – the Mininet and Cluster Monitor Tool available from https://git.opendaylight.org/gerrit/p/integration/test. It was clear that using Mininet was the only way to test the node functionality in the cluster in my case, because I did not have any hardware available that would support OpenFlow. However, as regards other tools, I found the Cluster Monitor Tool useful as long as it was easy to use and configure it, and it provided GUI interface and tracked the current states of nodes.

The Mininet will test that at least one node in the cluster operates normally, whereas the Cluster Monitor Tool will track the status of the nodes in the cluster. It will show two states – Leader and Follower. If a node is not running, an error message will be shown.

[4] Shard is a process that contains all the data in a module. Each instance of the cluster will have its own copies of the inventory data, the topology data and other necessary data required by the cluster.

The state Leader means that the node is responsible for this or another shard, while Follower is a backup state for a shard that is under the supervision of another node.

In order to test with the above mentioned Cluster Monitor Tool, it should be configured as shown in figure 18.

- Make another SSH connection to one of the controllers and clone test tools from git repository "`git clone https://git.opendaylight.org/gerrit/p/integration/test.git`". It contains ready-made scripts to test the OpenDaylight controller.
- Set IP addresses of the controllers in "*cluster.json"* file found in *"test/tools/clustering/cluster-monitor/"* directory. The *"user"* and *"pass"* field are default credentials of the controllers that should be the same. In order to change the controllers username or password, change credential values in *"users.properties"* file located in *"etc/"* folder of the OpenDaylight project [45].

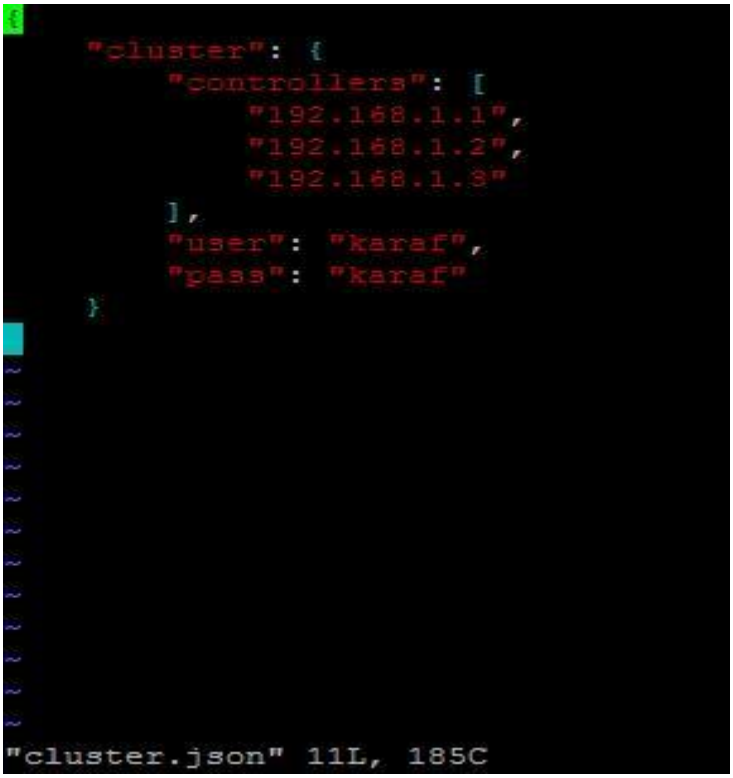

Figure 18 Cluster.json file configuration.

- Still in the same directory run *"monitor.py"* file. This window illustrates which shard the controller is responsible for. The shards' name on the X-axis and the

controllers' on the Y-axis will be shown. The *"Follower"* and the *"Leader"* are shown in the middle of the window as illustrated in figure 19. Thus, the status of the cluster can be monitored.

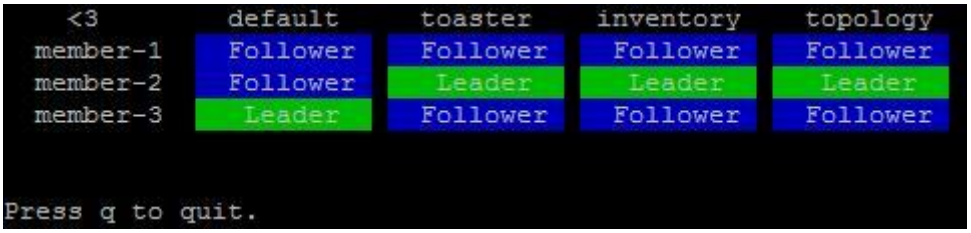

Figure 19 Cluster Monitor Tool.

Another option is to test the cluster using the Mininet network.

- Connect to the VM running Mininet.
- Execute "`sudo python multiple_ctrls.py`" script available from appendix 2 that was modified to meet my environment requirements.
- Test the connectivity with "`pingall`" command. If IP connectivity is successful, at least one controller in the cluster is operating normally (figure 20).

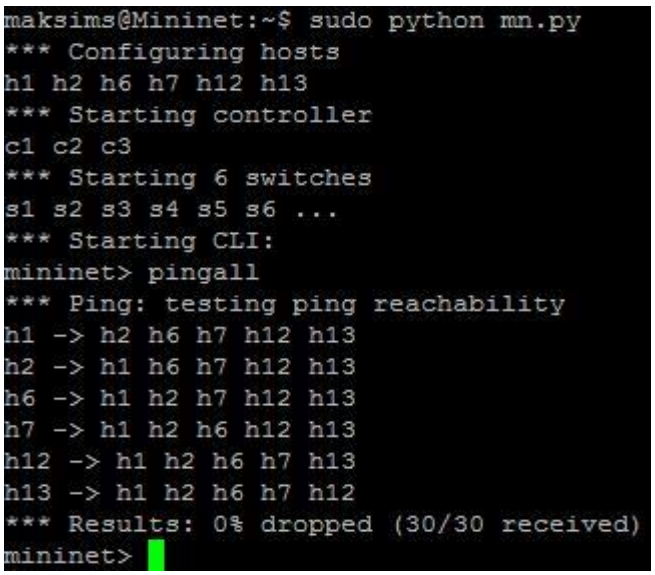

Figure 20 Mininet Script for Cluster Testing.

Even though, the clustering feature is very useful for enterprise usage, it still has drawbacks. For example, while running it, I faced unexpected behaviours when even a single node did not operate with Mininet. I had to reboot the whole VM and wait some time until it started to operate. My guess was that there were some components that could not be loaded due to the clustering features installed. At least log files stated this. Furthermore, clustering does not work with a number of applications available for the OpenDaylight project. To mention a few, those are Virtual Tenant Network (VTN), Defence4All and others.

4 Testing Application with OpenDaylight

This section covers installation and configuration of one example application available for the OpenDaylight project. As described in section 2, applications use the northbound APIs to communicate with underlying networking devices. They provide additional services using data collected by a controller.

I have chosen VTN as an example application, because its services and features are good examples of NVF. In addition, it does not require any special hardware for traffic analysis as the Defence4All firewall application and can be used with the Mininet network. But, as I said in section 3.5, there were some incompatibility issues while using some of the features and the clustering was among them. Hence, I had to remove cluster features to be able to use the VTN features.

4.1 Virtual Tenant Network

As an example application, Virtual Tenant Network (VTN) was used. VTN provides a possibility to create a multitenant network with an abstraction plane allowing users to define a L2/L3 network without knowing the physical network as shown in figure 21. [19.]

The Main benefit of the VTN is that users do not need to know the underlying infrastructure and details of configurations. Everything is abstracted by the VTN. Once the network is designed in the application, it will be automatically mapped into underlying network and then individual switches are configured leveraging the SDN control protocol. [27; 28.] Furthermore, the VTN does not set any special requirements for networking devices as long as it uses data collected by the controller and orchestrates through it utilizing a southbound protocol, for example, OpenFlow. [19; 28.]

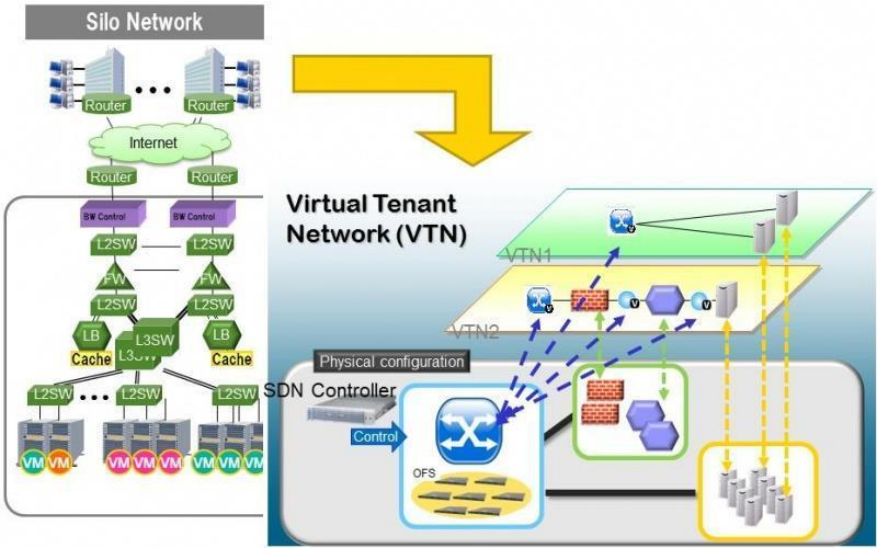

Figure 21 VTN Overview. Copied [5].

VTN is implemented with two components. One is the VTN manager, which is an OpenDaylight feature plugin that allows to configure components in OpenDaylight controller via REST interface. Another one is the VTN Coordinator, which is an external application that utilizes the interface pro-vided by the manager and provides a user with a VTN Virtualization. [28; 29; 30.]

Virtual Network Construction

In the VTN, a virtual network is constructed using virtual nodes (vBridge, vRouter) and virtual interfaces and links. It is possible to configure a network which has an L2 and L3 transfer function, by connecting the virtual interfaces made on virtual nodes via virtual links [19].

- vBridge: Represents L2 switch functions.
- vRouter: Provides router functions.
- vTep: Is a representation of Tunnel End Point (TEP).
- vTunnel: Is Tunnel logical representation.
- vBypass: Provides connectivity between controlled networks.
- Virtual interface: Represents virtual node's end points.
- Virtual Link (vLink): Is a connection between virtual interfaces. [19.]

Figure 22 shows an abstracted VTN network, where VRT is vRouter, BR1 and BR2 are vBridges. The vLinks connect all the components.

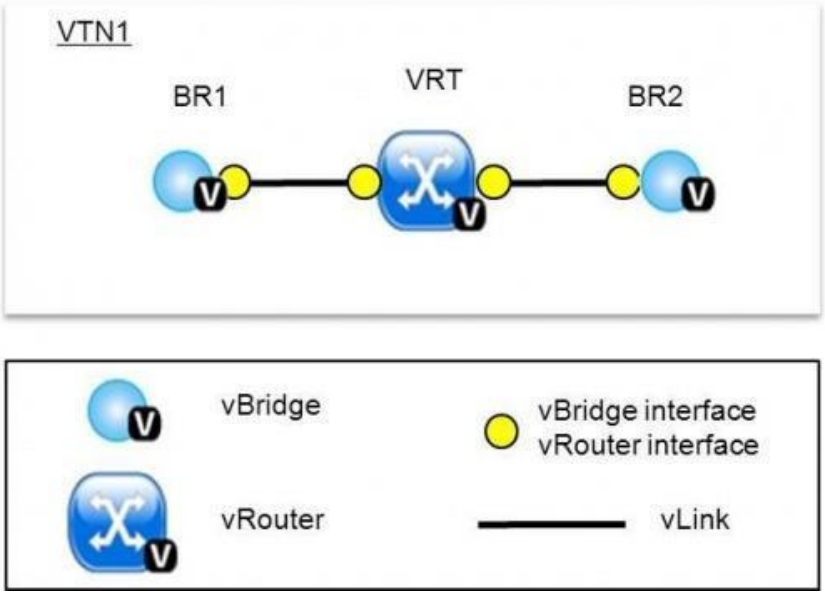

Figure 22 VTN Construction. Copied from [19].

To sum up, the VTN application enables users to configure the network without knowing the underlying infrastructure. Moreover, it is a tool that utilizes REST APIs for easier usage.

4.2 Environment Requirements

I had no special requirements for the application. It was installed on a separate preconfigured VM. The reason of having a separate VM was simple – I wanted to have a clearly defined VM that utilized its resource only for application-specific tasks. Thus, I was sure that there were no issues with resource allocation.

Another point to mention is that the application required some additional features to be installed in the controller distribution. But I encountered some restrictions while I was doing the project, but those should be vanished in next OpenDaylight release:

- VTN is not compatible with clustering mode.
- The application should be used without L2Switch project features installed. Otherwise the topology will be created automatically. The only way to use the VTN is by installing *"drop"* flow rules used for dropping packets specified in the flow tables. Other NFV features will be unavailable. [32.]

Thus, I recommend to create another fresh copy of the ODL controller at least without *"odl-mdsal-clustering"* features installed.

Furthermore, the VM, where applications will be installed, should have dependency packages preinstalled:

- Install Java JDK 7 and other dependencies through "`apt-get install`": "`gcc make g++ maven libboost-dev libcurl4-openssl-dev git pkg-config libjson0-dev libssl-dev unixodbc-dev ant xmlstarlet`". These packages are needed to compile the latest VTN application.
- Install "postgresql-9.3" through "`apt-get install`", which was required by the VTN application at the time of writing for storing its internal data: "`postgresql-9.3 postgresql-client-9.3 postgresql-client-com-mon postgresql-contrib-9.3 odbc-postgresql`". [19.]

4.3 Installing and Configuring a Virtual Tenant Network

As mentioned in section 4.1, the VTN has two components: VTN manager and VTN coordinator. The manager is a feature plugin installed on the controller, whereas the VTN coordinator is a standalone orchestration application.

VTN Manager

The VTN manager is installed in the same way as other features. In the *"karaf"* console execute "`feature:install odl-vtn-manager-rest`". This plugin provides REST APIs to internal components. The manager is ready to serve the VTN coordinator. [19; 27.]

VTN Coordinator

After all the dependency packages have been installed, it is time to install the latest VTN coordinator.

- Download the source code "`git clone https://git.opendaylight.org/gerrit/p/vtn.git`".

- Go to the *"vtn"* folder and check out a proper release. For the Lithium OpenDaylight controller do the following: "`git checkout stable/lithium`".
- Navigate to *"vtn/coordinator/"* directory.
- Execute "`mvn -f dist/pom.xml install`".
- To install the coordinator, extract a built program. It will be automatically placed in "/usr/local/bin/vtn" folder: "`tar -C/ -jxvf dist/target/distribution.vtn-coordinator-6.1.0.0-lithium-bin.tar.bz2`".

Before the VTN coordinator can be run, it will require some additional preparation:

- The coordinator is listening to port 8083 by default used by TomCat.
- To change the port, modify the "TOMCAT_PORT" in the "tomcat-env.sh" file found in *"/usr/local/vtn/tomcat/conf/"* directory.
- Set up the data base: *"/usr/local/vtn/sbin/db_setup".* The result is shown in figure 23.

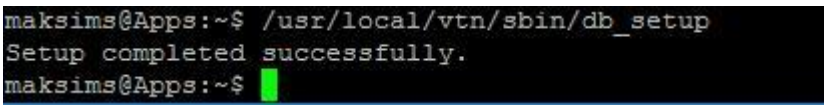

Figure 23 Setting up the Database.

The VTN coordinator can be started using the following command "`/usr/local/vtn/bin/vtn_start`". And stopped by "`/usr/local/vtn/bin/vtn_stop`".

In order to test if the coordinator has been properly installed and configured, the following REST command can be used (the VTN coordinator uses the IP address of the host machine):

```
curl --user admin:adminpass -H 'content-type: application/json' -X
\ GET 'http://[COORDINATOR_IP]:8083/vtn-webapi/api_version.json'
```

Listing 5. Get VTN version REST command.

As a result, the respond will show the installed version of the VTN: `{"api_version":{"version":"V1.2"}}`.[19.]

For security reasons, changing the VTN coordinator's password should have been done. Unfortunately, it could not be done with either REST APIs or with other methods for the Lithium release that was used in the project. [45.]

4.4 Configuring Layer2 Network with a Single Controller

After the VTN coordinator and Manager started to run, I began to test the system. First, I decided to test a simple L2 network configuration as shown in figure 24. The configuration was based on REST APIs provided by the VTN coordinator and utilized the following methods: "*GET", "POST", "PUT"* and *"DELETE"*.

First, I mapped the controller with the VTN, then VTN plane with vBridge was created. Afterwards, the physical interfaces of the hosts created by Mininet were mapped to the vBridge virtual interfaces. The entire code is available in appendix 3. The code was slightly modified to meet my environment requirements.

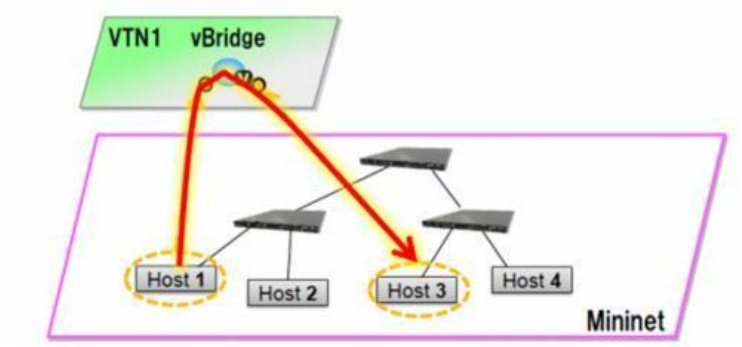

Figure 24 Single Controller Topology Example. Copied from [19].

To configure the Layer2 connectivity, proceed with the following steps:

- Create a topology using Mininet. The VTN will configure and establish connectivity between created hosts. "`sudo mn --controller=remote,ip=<controller-ip> --topo tree,2`".
- Create a Controller Information. This will map the controller instance with the VTN application.
- Create a VTN instance that will abstract the whole underlying infrastructure.
- Create a vBridge in the VTN that will connect hosts.
- Create two Interfaces into the vBridge.
- Get the list of logical ports configured.

- Configure two mappings on the interfaces. This will map logical interfaces with the virtual interfaces of the vBridge. [15.]

Next, try to ping hosts inside the Mininet network. The IP connectivity should be successful. To begin the next test, remove the VTN configuration.

4.5 Testing VLAN mapping

In this section VLAN mapping will be tested. There will be two VLANs (200 and 300), six hosts (three hosts in each VLAN) and three switches. The network topology is shown in figure 25.

As in section 4.4, the controller will be mapped with the VTN application and vBridges will have hosts mapped to their interfaces. Afterwards, the hosts will be put to different VLANs and mapped accordingly. The entire code is available in appendix 3, as slightly modified to meet the requirements.

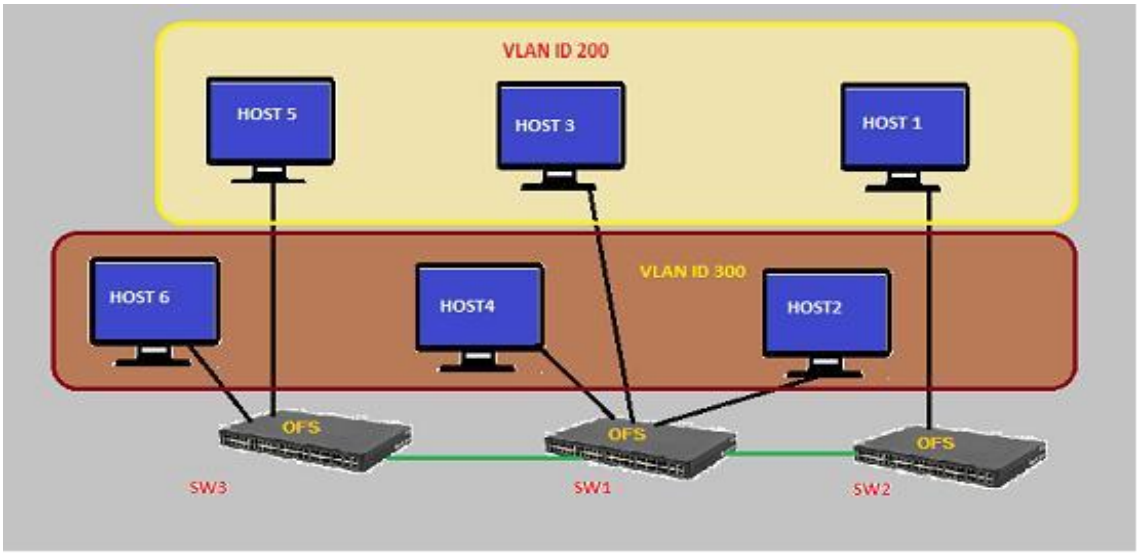

Figure 25 VLAN Map with VTN. Copied from [19].

To start testing VLAN mapping, do the following configurations and steps:

- Run the Mininet script that will create two VLANs (the script “multi_vlans.py” is available in appendix 2 originally available from opendaylight.org): “`sudo mn -`

-controller=remote,ip=192.168.64.13 --custom testing_vlans.py --topo mytopo".

- Create a Controller Information. This will map the controller instance with the VTN application.
- Create a VTN plane.
- Create a vBridge1 to be used as a gateway between hosts in VLAN 200.
- Create a vlan map with vlanid 200 for vBridge vBridge1.
- Create a vBridge2 for VLAN 300.
- Create a vlan map with vlanid 300 for vBridge vBridge2. [16.]

After the "pingall" command, the result should be the same as shown in figure 26.

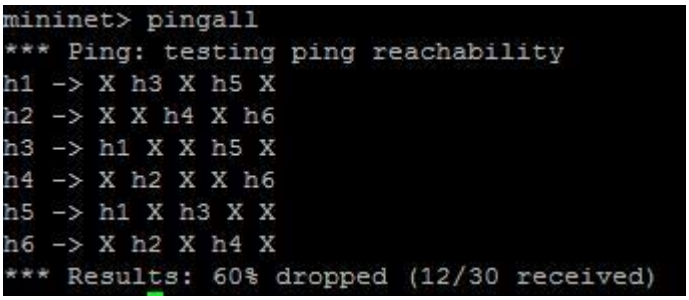

Figure 26 Ping Result.

The result in figure 26 means that IP connectivity between hosts in VLAN 200 as well as between hosts in VLAN 300 is successful, while there is no connectivity between the two VLAN groups.

This section concludes the practical part of the project. The OpenDaylight controller was tested with the VTN application using NVF possibilities.

5 Discussion

According to the results, the project reached its goals – the SDN system was built in a virtualized environment with some advanced features installed. The OpenDaylight controller, which was installed during the project, can be easily used using a virtualized network. Moreover, an example application, which was installed on a separate VM, was able to bring NFV features into the system.

Although, the OpenDaylight controller (Lithium version at the time of writing) also provides redundancy, its usage has limitations. It is compatible only with some of the features available for installation. Before using them, it is recommended to check at the wiki-pages[5] of features' limitations and problems that can be faced. For example, an example application, VTN, is incompatible with L2Switch and clustering projects. I had to create another copy of the controller in order to test the application.

Last, but not least is that there are still many issues with the controller implementation. Sometimes I could not make it work and error messages could not be fixed, because they required source code modification that was out of the scope of the project. Another example is the GUI of the controller. There are buttons and fields to install additional flows, but they do not operate at all. However, I tested the previous version of the controller called "Helium", and surprisingly, the mentioned features worked there. That proves the project was not mature at the time of writing.

[5] The OpenDaylight wiki-pages are available at https://wiki.opendaylight.org/

6 Conclusion

The goal of the project, to build an SDN system using OpenDaylight controller in a virtualized environment, was met. Along with the controller, an example application including virtual networking devices was installed. The system provided reliability and redundancy if properly used and the supported features were checked.

The OpenDaylight can be used as a main open-source controller if a user wants to create reliable, smart and modern networks providing that he or she is able to contribute to or make his or her own improvements to the source code.

As the final result, I gained experience with SDN technology and documented the project with installation instructions. That can be a basis for future Metropolia SDN offerings in terms of a practical learning system if students want to test and play with modern SDN technology. Future project improvements can be tested after the newest version called Berylium will be released. Those could be: VTN usage with a cluster of controllers and installing flows using DLUX GUI. Furthermore, if Metropolia has hardware with OpenFlow support, an SDN system could be integrated into its campus infrastructure.

References

1. Nadeu TD, Gray K. SDN: Software Defined Networks.
   Sebastopol, CA: O'Reilly; 2013.

2. Isolani PH, Wickboldt JA, Both CB, Rochol J, Granville LZ. Interactive monitoring, visualization, and configuration of OpenFlow-based SDN [online]. Porto Alegre, Brazil: IEEE; May 2015. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7140294&newsearch=true&queryText=Interactive%20monitoring,%20visualization,%20and%20configuration%20of%20OpenFlow-based%20SDN. Accessed 10 February 2016.

3. SDX Central. Understanding the SDN Architecture [online]. Sunnyvale, CA: SDxCentral. URL: https://www.sdxcentral.com/resources/sdn/inside-sdn-architecture/. Accessed 10 February 2016.

4. Nishtha, Manu Sood. Software defined network – Architectures [online]. Shimla, India: IEEE; December 2014. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7030788&newsearch=true&queryText=Software%20Defined%20Network%20-%20Architectures. Accessed 10 February 2016.

5. OpenDaylight Community. OpenDaylight Web Page [online].
   URL: http://opendaylight.org/. Accessed 1 March 2016.

6. Internet Research Task Force. Software-Defined Networking (SDN): Layers and Architecture Terminology [online]. January 2015.
   URL: https://tools.ietf.org/html/rfc7426/. Accessed 11 February 2016.

7. Internet Research Task Force. The Open vSwitch Database Management Protocol [online]. December 2013.
   URL: https://tools.ietf.org/html/rfc7047/. Accessed 11 February 2016.

8. OpenDaylight community. OpenDaylight Lisp Flow Mapping:User Guide for Hydrogen [online].
   URL: https://wiki.opendaylight.org/view/OpenDaylight_Lisp_Flow_Mapping:Architecture/. Accessed 11 February 2016.

9. Jarschel M, Oechsner S, Schlosser D, Pries R, Goll S, Tran-Gia P. Modeling and performance evaluation of an OpenFlow architecture [online]. San Francisco, CA: IEEE; September 2011. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6038457&newsearch=true&queryText=9.%09Modeling%20and%20performance%20evaluation%20of%20an%20OpenFlow%20architecture%20/. Accessed 14 February 2016.

10. OpenDaylight community. Running and testing an OpenDaylight Cluster [online]. URL: https://wiki.opendaylight.org/view/Running_and_testing_an_OpenDaylight_Cluster/. Accessed 19 February 2016.

11. McKeown N, Parulkar G, Shenker S, Anderson T, Peterson L, Turner J, Balakrishnan H, Rexford J. OpenFlow: Enabling Innovation in Campus Networks [online]. 14 March 2008. URL: http://archive.openflow.org/documents/openflow-wp-latest.pdf. Accessed 14 February 2016.

12. Open Networking Foundation. OpenFlow Switch Specification [online]. 14 October 2013. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf. Accessed 14 February 2016.

13. SDX Central. What are SDN Northbound APIs? [online]. Sunnyvale, CA: SDxCentral. URL: https://www.sdxcentral.com/resources/sdn/north-bound-interfaces-api/. Accessed 14 February 2016.

14. OpenDaylight community. OpenDaylight DLUX:DLUX Karaf Feature [online]. URL: https://wiki.opendaylight.org/view/OpenDaylight_DLUX:DLUX_Karaf_Feature/. Accessed 19 February 2016.

15. OpenDaylight community. How to configure L2 Network with Single Controller [online]. URL: https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_(VTN):VTN_Coordinator:Release/Lithium/VTN/How-Tos/How_To_configure_L2_Network_with_Single_Controller/. Accessed 19 February 2016.

16. OpenDaylight community. How to test vlan-map in Mininet environment [online]. URL: https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_(VTN):VTN_Coordinator:RestApi:How_to_test_vlan-map_in_Mininet_environment/. Accessed 19 February 2016.

17. Li L, Wu Chou, Wei Zhou, Min Luo. Design Patterns and Extensibility of REST API for Networking Applications [online]. IEEE; 11 January 2016. URL:http://ieeexplore.ieee.org/xpl/articleDetails.jsp?ar-number=7378522&newsearch=true&queryText=17.%09Design%20Patterns%20and%20Extensibility%20of%20REST%20API%20for%20Networking%20Applications%20/. Accessed 14 February 2016.

18. Medvedev J, Tkacik A, Varga R, Gray K. OpenDaylight: Towards a Model-Driven SDN Controller Architecture [online]. Sydney, NSW: IEEE; 19 June 2014. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?ar-number=6918985&newsearch=true&queryText=OpenDaylight:%20Towards%20a%20Model-Driven%20SDN%20Controller%20architecture/. Accessed 14 February 2016.

19. OpenDaylight community. Release/Lithium/VTN/User Guide [online]. URL: https://wiki.opendaylight.org/view/Release/Lithium/VTN/User_Guide/. Accessed 19 February 2016.

20. OpenDaylight community. L2 Switch:Helium:User Guide [online]. URL: https://wiki.opendaylight.org/view/L2_Switch:Helium:User_Guide/. Accessed 19 February 2016.

21. OpenDaylight community. BGP LS PCEP [online].
URL: https://wiki.opendaylight.org/view/BGP_LS_PCEP:Main/.
Accessed 1 March 2016.

22. Lightbend. Persistence [online].
URL: http://doc.akka.io/docs/akka/snapshot/java/persistence.html/.
Accessed 1 March 2016.

23. Lightbend. Cluster [online].
URL: http://doc.akka.io/docs/akka/snapshot/common/cluster.html#cluster/.
Accessed 1 March 2016.

24. Lightbend. Remoting [online]. URL: http://doc.akka.io/docs/akka/snapshot/java/remoting.html/. Accessed 1 March 2016.

25. OpenDaylight community. OpenDaylight Controller:MD-SAL:Architecture:Clustering [online]. URL: https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Archi-tecture:Clustering/. Accessed 15 February 2016.

26. Roland Huß. Jolokia [online]. 16 February 2016. URL: https://jolokia.org/features-nb.html/. Accessed 1 March 2016.

27. OpenDaylight community. Release/Lithium/VTN/Installation Guide [online].
URL: https://wiki.opendaylight.org/view/Release/Lithium/VTN/Installation_Guide. Accessed 14 February 2016.

28. OpenDaylight community. OpenDaylight Virtual Tenant Network (VTN):Implementation [online]. URL: http://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Net-work_(VTN):Implementation/. Accessed 1 March 2016.

29. OpenDaylight community. OpenDaylight Virtual Tenant Network (VTN):VTN Coordinator [online]. URL: https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Net-work_(VTN):VTN_Coordinator/. Accessed 1 March 2016.

30. OpenDaylight community. OpenDaylight Virtual Tenant Network (VTN):VTN Manager [online]. URL:https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_(VTN):VTN_Manager/. Accessed 1 March 2016.

31. OpenDaylight community. Getting and Installing OpenDaylight [online]. URL: https://www.opendaylight.org/installing-opendaylight/. Accessed 1 March 2016.

32. OpenDaylight community. OpenDaylight Virtual Tenant Network (VTN):Beryllium Release Plan [online].
URL: https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_(VTN):Beryllium_Release_Plan/. Accessed 1 March 2016.

33. SDX Central. What is NFV – Network Functions Virtualization – Definition? [online]. Sunnyvale, CA: SDxCentral. URL: https://www.sdxcentral.com/resources/nfv/whats-network-functions-virtu-alization-nfv/. Accessed 1 March 2016.

34. SDX Central. What is White Box Switching and White Box Switches? [online]. Sunnyvale, CA: SDxCentral. URL: https://www.sdxcentral.com/resources/whitebox/what-is-white-box-networking/. Accessed 1 March 2016.

35. SDX Central. What are SDN Southbound APIs? [online]. Sunnyvale, CA: SDxCentral. URL: https://www.sdxcentral.com/resources/sdn/southbound-interface-api/. Accessed 1 March 2016.

36. OpenDaylight community. OpenDaylight Controller:MD-SAL:Architecture [online]. URL: https://wiki.opendaylight.org/view/OpenDaylight_Controller:MD-SAL:Architecture/. Accessed 1 March 2016.

37. OpenDaylight community. OpenDaylight Controller:Config:config.ini [online]. URL:https://wiki.opendaylight.org/view/OpenDaylight_Controller:Config:config.ini/. Accessed 1 March 2016.

38. OpenDaylight community. L2 Switch:Helium:Developer Guide [online]. URL: https://wiki.opendaylight.org/view/L2_Switch:Helium:Developer_Guide/. Accessed 1 March 2016.

39. OpenDaylight community. L2 Switch:Helium:User Guide [online]. URL: https://wiki.opendaylight.org/view/L2_Switch:Helium:User_Guide/. Accessed 1 March 2016.

40. Mininet Team. Documentation [online]. 13 Novermber 2015. URL: https://github.com/mininet/mininet/wiki/Documentation/. Accessed 1 March 2016.

41. Mininet Team. Mininet Walkthrough [online]. URL: http://mininet.org/walkthrough/. Accessed 1 March 2016.

42. OpenDaylight community. AAA:Changing Account Passwords [online]. URL: https://wiki.opendaylight.org/view/AAA:Changing_Account_Passwords/. Accessed 1 March 2016.

43. OpenDaylight community. Community Labs [online]. URL: https://www.opendaylight.org/community-labs/. Accessed 8 March 2016.

44. Chandrasekar Kannan. Running OpenDaylight [online]. CloudRouter; 12 November 2015. URL: https://cloudrouter.atlassian.net/wiki/dis-play/CPD/Running+OpenDaylight/. Accessed 8 March 2016.

45. OpenDaylight community. VTN Coordinator:RestApi: User Function [online]. URL: https://wiki.opendaylight.org/view/OpenDaylight_Virtual_Tenant_Network_(VTN):VTN_Coordinator:RestApi:_User_Function/. Accessed 8 March 2016.

46. OpenFlow [online]. OpenFlow Web Page. URL: http://openflow.com/. Accessed 17 February 2015.

47. OpenDaylight Community [online]. OpenDaylight Wiki. URL: http://wiki.opendaylight.org/. Accessed 10 January 2015.

Virtual Machines' Interface Configurations

Controller-1

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static address
        10.94.159.100        net-
        mask      255.255.255.0
        gateway 10.94.159.254
        dns-nameservers 10.94.1.4

auto eth1
iface eth1 inet static address
        192.168.1.1     netmask
        255.255.255.0
```

Controller-2

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static address
        10.94.159.101        net-
        mask      255.255.255.0
        gateway 10.94.159.254
        dns-nameservers 10.94.1.4

auto eth1
iface eth1 inet static address
        192.168.1.2     netmask
        255.255.255.0
```

Controller-3

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static address
        10.94.159.104        net-
        mask      255.255.255.0
        gateway 10.94.159.254
        dns-nameservers 10.94.1.4

auto eth1
iface eth1 inet static address
        192.168.1.3     netmask
        255.255.255.0
```

Mininet

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static address
        10.94.159.102        net-
        mask      255.255.255.0
        gateway 10.94.159.254
        dns-nameservers 10.94.1.4

auto eth1
iface eth1 inet static address
        192.168.1.11   netmask
        255.255.255.0

auto eth2
iface eth2 inet static address
        1.1.1.1            netmask
        255.255.255.0
```

Applications

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static address
        10.94.159.103        net-
        mask      255.255.255.0
        gateway 10.94.159.254
        dns-nameserver 10.94.1.4

auto eth1
iface eth1 inet static address
        192.168.1.12   netmask
        255.255.255.0
```

Mininet Network Configuration Scripts

Script multiple_ctrls.py

```
#!/usr/bin/python

from mininet.cli import CLI
from mininet.node import Controller, OVSKernelSwitch, RemoteController
from mininet.log import setLogLevel, info
from mininet.net import Mininet


def emptyNet():

   net = Mininet(controller=RemoteController, switch=OVSKernelSwitch)

   c1 = net.addController('c1', controller=RemoteController, ip="192.168.1.1",
port=6633)
   c2 = net.addController('c2', controller=RemoteController, ip="192.168.1.2",
port=6633)
   c3 = net.addController('c3', controller=RemoteController, ip="192.168.1.3",
port=6633)

   h1 = net.addHost( 'h1', ip='10.0.0.1' )
   h2 = net.addHost( 'h2', ip='10.0.0.2' )
   h6 = net.addHost( 'h6', ip='10.0.1.6' )
   h7 = net.addHost( 'h7', ip='10.0.1.7' )
   h12 = net.addHost( 'h12', ip='10.0.3.12' )
   h13 = net.addHost( 'h13', ip='10.0.4.13' )

   s1 = net.addSwitch( 's1' )
   s2 = net.addSwitch( 's2' )
   s3 = net.addSwitch( 's3' )
   s4 = net.addSwitch( 's4' )
   s5 = net.addSwitch( 's5' )
   s6 = net.addSwitch( 's6' )


   s4.linkTo( s6 )
   s5.linkTo( s4 )
   s6.linkTo( s3 )

   s1.linkTo( s3 )
   s2.linkTo( s4 )
   s3.linkTo( s5 )

   s1.linkTo( h1 )
   s1.linkTo( h2 )

   s2.linkTo( h6 )
   s2.linkTo( h7 )
```

```
s3.linkTo( h12 )
    s3.linkTo( h13 )

    net.build()
    c2.start()
    c1.start()
    c3.start()
    s1.start([c1,c2,c3])
    s2.start([c1,c2,c3])
    s3.start([c1,c2,c3])
    s4.start([c1,c2,c3])
    s5.start([c1,c2,c3])
    s6.start([c1,c2,c3])

    net.start()
    net.staticArp()
    CLI(     net     )
    net.stop()
if __name__ == '__main__':
     setLogLevel( 'info' ) emp-
     tyNet()
```

Script multi_vlan.py

```
#!/usr/bin/python

from mininet.node import Host, RemoteController
from mininet.topo import Topo
import apt

#Note Vlan package check only work with ubuntu
#Please comment the package check if your running the script other than ubuntu

#package check Start
cache = apt.Cache()
if  cache['vlan'].is_installed:
    print "Vlan installed"
else:
    print "ERROR:VLAN package not installed please run sudo apt-get install vlan"
    exit(1)
#package check End

class VLANHost( Host ):
        def config( self, vlan=100, **params ):
                """Configure VLANHost according to (optional) parameters:
                        vlan: VLAN ID for default interface"""
                r = super( Host, self ).config( **params )
                intf = self.defaultIntf()
# remove IP from default, "physical" interface
                self.cmd( 'ifconfig %s inet 0' % intf )
# create VLAN interface
                self.cmd( 'vconfig add %s %d' % ( intf, vlan ) )
# assign the host's IP to the VLAN interface
                self.cmd( 'ifconfig %s.%d inet %s' % ( intf, vlan, params['ip'] ) )
# update the intf name and host's intf map
                newName = '%s.%d' % ( intf, vlan )
# update the (Mininet) interface to refer to VLAN interface name
                intf.name = newName
# add VLAN interface to host's name to intf map
                self.nameToIntf[ newName ] = intf re-
                turn r

class MyTopo( Topo ): "Simple
    topology    example."    def
    __init__( self ):
        "Create custom topo."
        # Initialize   topology
        Topo.__init__( self )
        # Add hosts and switches
        host1=self.addHost( 'h1', cls=VLANHost, vlan=200)
        host2=self.addHost( 'h2', cls=VLANHost, vlan=300)
        host3=self.addHost( 'h3', cls=VLANHost, vlan=200)
        host4=self.addHost( 'h4', cls=VLANHost, vlan=300)
        host5=self.addHost( 'h5', cls=VLANHost, vlan=200)
        host6=self.addHost( 'h6', cls=VLANHost, vlan=300)
```

```
        s1 = self.addSwitch( 's1' )
        s2 = self.addSwitch( 's2' )
        s3 = self.addSwitch( 's3' )

        self.addLink(s1,host1)
        self.addLink(s1,s2)
        self.addLink(s2,host2)
        self.addLink(s2,host3)
        self.addLink(s2,host4)
        self.addLink(s1,s3)
        self.addLink(s3,host5)
        self.addLink(s3,host6)

topos = { 'mytopo': ( lambda: MyTopo() ) } [16.]
```

VTN Configuration Scripts

Configuring L2 connectivity with single controller

- Create a Controller.

```
curl -v --user admin:adminpass -H 'content-type: application/json' -X POST -d '{"controller": {"controller_id": "controllerone", "ipaddr":"[controller-ip]", "type": "odc", "version": "1.0", "auditstatus":"enable"}}' http://[coordinator-ip]:8083/vtn-webapi/controllers.json
```

- Create a VTN.

```
curl -v --user admin:adminpass -H 'content-type: application/json' -X POST -d '{"vtn" : {"vtn_name":"vtn1","description":"test VTN" }}' http://[coordinator-ip]:8083/vtn-webapi/vtns.json
```

- Create a vBridge in the VTN.

```
curl -v --user admin:adminpass -H 'content-type: application/json' -X POST -d '{"vbridge" : {"vbr_name":"vBridge1","controller_id":"controllerone","domain_id":"(DEFAULT)" }}' http://[coordinator-ip]:8083/vtn-webapi/vtns/vtn1/vbridges.json
```

- Create two Interfaces into the vBridge.

```
curl -v --user admin:adminpass -H 'content-type: application/json' -X POST -d '{"interface": {"if_name": "if1","description": "if_desc1"}}' http://[coordinator-ip]:8083/vtn-webapi/vtns/vtn1/vbridges/vBridge1/interfaces.json
```

```
curl -v --user admin:adminpass -H 'content-type: application/json' -X POST -d '{"interface": {"if_name": "if2","description": "if_desc2"}}' http://[coordinator-ip]:8083/vtn-webapi/vtns/vtn1/vbridges/vBridge1/interfaces.json
```

- Get the list of logical ports configured.

```
curl -v --user admin:adminpass -H 'content-type: application/json' -X GET http://[coordinator-ip]:8083/vtn-webapi/controllers/controllerone/do-mains/\(DEFAULT\)/logical_ports.json
```

- Configure two mappings on the interfaces.

```
curl -v --user admin:adminpass -H 'content-type: application/json' -X PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:3-s3-eth1"}}' http://[coordinator-ip]:8083/vtn-webapi/vtns/vtn1/vbridges/vBridge1/interfaces/if1/portmap.json
```

```
curl –v --user admin:adminpass -H 'content-type: application/json' -X PUT -d '{"portmap":{"logical_port_id": "PP-OF:openflow:2-s2-eth1"}}' http://[coordinator-ip]:8083/vtn-webapi/vtns/vtn1/vbridges/vBridge1/interfaces/if2/portmap.json
```

[15.]

Configuring L2 Connectivity with VLANs

- Create a controller

```
curl -v --user admin:adminpass -H 'content-type: application/json' -X POST -d '{"controller": {"controller_id": "controllerone", "ipaddr":"[controller-ip]", "type": "odc","version": "1.0", "auditstatus":"enable"}}' http://[coordinator-ip]:8083/vtn-webapi/controllers.json
```

- Create a VTN

```
curl -v -X POST -H 'content-type: application/json' -H 'username: admin' -H 'password: adminpass' -d '{"vtn" : {"vtn_name":"vtn1","description":"test VTN" }}' http://[coordinator-ip]:8083/vtn-webapi/vtns.json
```

- Create a vBridge(vBridge1)

```
curl -v -X POST -H 'content-type: application/json' -H 'username: admin' -H 'password: adminpass' -d '{"vbridge" : {"vbr_name":"vBridge1","controller_id":"controllerone","domain_id":"(DEFAULT)"}}' http://[coordinator-ip]:8083/vtn-webapi/vtns/vtn1/vbridges.json
```

- Create a vlan map with vlanid 200 for vBridge vBridge1

```
curl -v -X POST -H 'content-type: application/json' -H 'username: admin' -H 'password: adminpass' -d '{"vlanmap" : {"vlan_id": 200 }}' http://[coordinator-ip]:8083/vtn-webapi/vtns/vtn1/vbridges/vBridge1/vlanmaps.json
```

- Create a vBridge (vBridge2)

```
curl -v -X POST -H 'content-type: application/json' -H 'username: admin' -H 'password: adminpass' -d '{"vbridge" : {"vbr_name":"vBridge2","controller_id":"controllerone","domain_id":"(DEFAULT)"}}' http://[coordinator-ip]:8083/vtn-webapi/vtns/vtn1/vbridges.json
```

- Create a vlan map with vlanid 300 for vBridge vBridge2

```
curl -v -X POST -H 'content-type: application/json' -H 'username: admin' -H 'password: adminpass' -d '{"vlanmap" : {"vlan_id": 300 }}' http://[coordinator-ip]:8083/vtn-webapi/vtns/vtn1/vbridges/vBridge2/vlanmaps.json
```

[16.]